INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Correctness of Java Card Tokenisation

Ewen Denney

N°3831

Décembre 1999

THÈME 2

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--3831--FR+ENG

Correctness of Java Card Tokenisation

Ewen Denney

Thème 2 — Génie logiciel
et calcul symbolique
Projet Lande

Rapport de recherche n° 3831 — Décembre 1999 — 34 pages

Abstract: We present a formalisation of the bytecode optimisation of Sun's Java Card language from the class file to CAP file format as a set of constraints between the two formats and define and prove its correctness. Java Card bytecode is formalised as an abstract operational semantics, which can then be instantiated into the two formats. The optimisation is given as a logical relation such that the instantiated semantics are observably equal.

Key-words: Java Card, bytecode, optimisation, logical relations, data refinement

(Résumé : tsvp)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

La correction de la tokenisation de Java Card

Résumé : Nous présentons une formalisation de l'optimisation de bytecode du langage Java Card de Sun qui est réalisé par une transformation du format "class file" au format "CAP file", La formalisation s'exprime comme un ensemble de contraintes entre les deux formats. Le bytecode de Java Card est formalisé comme une sémantique opérationnelle abstraite, qui peut alors être instanciées dans les deux formats. L'optimisation est donnée comme une relation logique telle que les sémantiques instanciées sont observablement égales.

Mots-clé : Java Card, bytecode, optimisation, relations logiques, raffinement de données

The Java Card language [Sun97] is a trimmed down dialect of Java aimed at programming smart cards. As with Java, Java Card is compiled into bytecode, which is then verified and executed on a virtual machine [LY97], installed on a chip on the card itself. However, the memory and processor limitations of smart cards necessitate a further stage, in which the bytecode is optimised from the standard class file format of Java, to the *CAP file* format [Sun99]. The core of this optimisation is a *tokenisation* in which names are replaced with tokens enabling a more direct lookup of various entities.

We describe a semantic framework for proving the correctness of Java Card tokenisation. The basic idea is to give an abstract description of the constraints given in the official specification of the tokenisation and show that any transformation satisfying these constraints is 'correct'. This is independent of showing that there actually exists a collection of functions satisfying these constraints. This report concentrates on proving the correctness of the specification. The formal development of an algorithm will be the subject of another report.

The main advantage of decoupling 'correctness' into two steps is that we get a more general result. Rather than proving the correctness of one particular algorithm, we are able to show that the constraints described in Sun's official specification [Sun99] (given certain assumptions) are sufficient. Moreover, the technique used to develop an algorithm is orthogonal to this proof.

1 The Conversion

We give a brief sketch of the transformation process. We assume that the reader has a basic understanding of the various elements of the Java Virtual Machine and class file format.

Java source code is compiled on a class by class basis into the *class file* format. By contrast, Java Card *CAP files* correspond to packages. They are produced by the *conversion* of a collection of class files. In fact, the conversion process also takes a number of *export files* as input, but we will ignore these here. Indeed, this is just one of several simplifying assumptions we make.

The 'transformation' is presented in [Sun99] as a collection of constraints on the CAP file, rather than as an explicit correspondence between class and CAP formats. Instead, we adopt a simplified definition of the transformation, only considering classes, constant pools, fields and methods. In particular, we ignore exceptions and interfaces.

In the class file format, methods, fields and so on are referred to using a certain naming convention. In CAP files, instead, tokens are ascribed to the various entities. The idea is that if a method, say, is publically visible[1], then it is ascribed a token. If the method is only visible within its package, then it is referred to directly using an offset into the relevant data structure. Thus references are either internal or external. In addition, 'top-level' references, to packages (and applets) are made using *application identifiers* (AIDs).

CAP files consist of a number of components, of which we will consider the constant pool, class, method, static field and descriptor components. One significant difference between the two formats is the way in which the method tables are arranged. In a class file, the methods item contains all the information relevant to methods defined in that class. In the CAP file, this information is shared between the class and method components. The method component contains the implementation details (*i.e.* the bytecode) for the methods defined in this package. The class component is a collection of class information structures. Each of these contains separate tables for the package and public methods, mapping tokens to offsets into the method component. The method tables contain the information necessary for resolving any method call in that class. If a class inherits a method from a superclass then it may be that the method token is included in the relevant table, or that the table of the superclass should be searched. There is a choice, therefore, between copying all inherited methods, or having a more compressed table. The specification does not constrain this choice.

Another optimisation concerns method references. These are tagged to indicate whether they correspond to the call of a 'supermethod', that is, the method of a superclass. This comes from using the `super` keyword in the source code (thus avoiding overriding). Retaining this information in the bytecode allows a more efficient location of the information in the tables.

What we have described are those aspects concerned with the rearrangement into CAP format. There are also a number of mandatory optimisations such as the inlining of final fields, and the type-based specialisation of instructions. The order of these stages is not specified. Indeed, a converter is at liberty to implement further optimisations.

Most of the work in the proof lies in the various definitions: defining the semantics of the virtual machine independently of the underlying format, and formalising the main stages of the transformation. Given this

[1] We follow the terminology of [Sun99], where a method is *public visible* if it has either a `protected` or a `public` modifier, and *package visible* if it is declared `private` or has no visibility modifier.

framework, the proof, as such, is relatively small. It is natural when proving correctness to consider all the transformation steps simultaneously. The modularity is in the definition. Thus, by "Java card tokenisation" we mean both the assignment of tokens to the various named items, and the rearrangement in the CAP file format. We also take the compression of method tables into account.

There is a more detailed discussion of the differences between Java and Java Card in the official documentation [Sun97, Sun99].

2 Related Work

There have been a number of formalisations of the Java Virtual Machine which have some relevance for our work here on Java Card. Bertelsen [Ber97] gives an operational semantics which we have used as a starting point. He also considers the verification conditions, which considerably complicates the rules, however. Börger and Schulte [BS98] have a different approach, though also make use of auxiliary functions to a certain extent. Pusch has formalised the JVM in HOL [Pus98]. Like us, she considers the class file to be well-formed so that the hypotheses of rules are just assignments. The operational semantics is presented directly as a formalisation in HOL, whereas we have chosen (equivalently) to use inference rules. All these works make various simplifications and abstractions. However, since these are formalisations of Java rather than Java Card they do not consider the CAP file format.

In contrast, the work of Lanet and Requet [LR98] is specifically concerned with Java Card. They also aim to prove the correctness of Java Card tokenisation. Their work can be seen as complementing ours. They concentrate on optimisations, including the type specialisation of instructions, and do not consider the conversion as such. In contrast, this is an aspect which we have ignored completely but have, however, specified the conversion. Their formalism is based on the B method, so the specification and proof are presented as a series of refinements.

In [Pus96], Pusch proves the correctness of an implementation of Prolog on an abstract machine, the WAM. The proof structure is similar to ours, although there are refinements through several levels. There are operational semantics for each level, and correctness in expressed in terms of equivalence between levels. The differences between the semantics are significant, since they are not factored out into auxiliary functions as here. She uses a big-step operational semantics, which is not appropriate for us because we wish to compare intermediate states. Moreover, she uses an abstraction function on the initial state, the results being required to be identical, whereas we have a *relation* for both initial and final states.

3 Overview of Formalisation

We will present the transformation from class file to CAP file as a transformation of virtual machines, that is, from the JVM to the JCVM. Since Java Card is a sublanguage of Java it can be executed on a JVM, although the intention is that the conversion is an integral part of the compilation process, and that only the CAP file is executed.

The first issue to be addressed is determining in what sense, exactly, the conversion to token format should be regarded as an equivalence. We cannot, for example, simply say that the JVM and JCVM have the same behaviour for all bytecodes, in class and CAP file format respectively, because, *a priori*, the states of the virtual machines are themselves in different formats.

We adopt a simple form of equivalence based on the notion of *representation independence* [Mit96]. This is expressed in terms of so-called *observable* types. This limits us to comparing the two interpretations in terms of words (there are no double words in Java Card), but this is sufficient to observe the operand stack and local variables.

Representation independence may be proven by defining *coupling relations* between the two formats, which respect the tokenisation and are the identity at observable types. This can be seen as formalising a *data refinement* from class to CAP files.

We formalise the relations nondeterministically as any family of relations which satisfies certain constraints, rather than as explicit transformations. This is because there are many possible tokenisations and we wish to prove any reasonable optimisation correct. Formally, we say that a function is representation independent if it maps related inputs to related outputs. This is the definition of a *logical relation* at function types.

We follow numerous researchers in this area and formalise the virtual machines in an operational style, as transition relations over abstract machines. We adopt the action semantics formalism of Mosses [Mos98].

This is convenient as by presenting the bytecode semantics in a modular manner we can more easily make the comparison between the two formats where significant. We prove the correctness of tokenisation with respect to these semantics. However, the particular formalisation of the semantics is orthogonal to the technique used for proving equivalence. The main point is to give a set of operational rules which can be used for both virtual machines, with all the semantic differences abstracted out into a number of auxiliary functions.

The semantics is given in a mixture of operational and denotational styles. We formalise the JCVM operationally, parameterised with respect to a number of auxiliary functions which are then interpreted denotationally.

To illustrate how it is natural to conceive the operational semantics independently of certain auxiliary functions, we consider dynamic method lookup, used in the semantics of the method invocation instructions. The lookup function which searches for the implementation of a method is dependent on the layout of the method tables. There are also a number of choices for how it is affected by method modifiers, each of which is apparently consistent with the official specification. The operational rules giving the semantics of the method invocation instructions, presented in Section 5.1, are parameterised with respect to the lookup function. Then in Sections 6 and 7 two possible interpretations of lookup (and the other auxiliary functions) are given. A further choice would be to give an abstract interpretation to the auxiliary functions or, going in the opposite direction, to include error information. For example, if the bytecode is not assumed to be verified, the lookup function could return `NoSuchMethodError` or `IllegalAccessError`.

Although the equivalence of dynamic method lookup could be regarded as the aim of the proof, in fact 'correctness' is distributed throughout all of the transformation and equivalence of lookup uses equivalence of the transformation of classes, and so on.

The operational semantics, together with the interpretations of the auxiliary functions, induces an 'interpretation' of the bytecode, and it is in terms of this that we compare the two formats. In this spirit, we will use 'name interpretation' and 'token interpretation' to refer to the semantics of the JVM and JCVM respectively.

Hence we consider the tokenisation to be correct if *we can give a family of coupling relations which respects the tokenisation, and which is the identity on observable types.* We set this as the goal of the proof. Here we will just be concerned with the correctness of the tokenisation and conversion to CAP format, and not any inlining or further optimisation that might take place.

We give types to the various entities converted during tokenisation, such as `Class_ref` and `Constant_pool`. We include a type, `Bytecode`, since the bytecode itself changes during tokenisation. This is due, amongst other reasons, to the presence of constant pool indices as arguments to instructions.

Let us write $[\![.]\!]_{name}$ for the name interpretation (class format), and $[\![.]\!]_{tok}$ for the token interpretation (CAP format). We interpret both types and auxiliary functions. For example

$$[\![\texttt{Method\_ref}]\!]_{name} = \texttt{Class\_name} \times \texttt{Method\_name} \times \texttt{Type}$$

$$[\![\texttt{Method\_ref}]\!]_{tok} = \texttt{Class\_ref} \times \texttt{Method\_token}$$

The lookup function

$$\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \to \texttt{Class\_ref} \times \texttt{Bytecode}$$

is interpreted, in turn, as the two functions defined below. Then for each type, θ, we define a relation $R_\theta \subseteq [\![\theta]\!]_{name} \times [\![\theta]\!]_{tok}$

One technical problem is that some types are most naturally considered as being local to a certain context, or dependent on another type. For example, class references in the token interpretation can either be external to a package or internal. In the second case, they are given as an offset which does not make sense out of the package. Thus $R_{\texttt{Class\_ref}}$ must relate class names to both internal and external token references. Another example is that constant pool indices in the bytecode are assumed to have a label indicating the relevant constant pool (whereas, in reality, this is evident from the context.) We can get round this by assuming that data is paired with something to indicate its context whenever necessary. A more elegant approach would use dependent types.

We start from the observation that not all instructions use the heap or the environment. It is these which are sensitive to the alterations in the layout of the constant pool and class hierarchy, which take place during tokenisation. Thus we will ignore the instructions concerned with immediate operations, stack manipulation, local variables, and branching.[2]

[2]In fact, we will not consider the array access instructions either, one of which (`aastore`) accesses the environment, but we could easily make this extension.

Secondly, since the bytecode is assumed to have been verified (and is not self-modifying), we can regard it as having been assembled into an abstract syntax. This simplifies the presentation and lets us abstract away from details of program counters and use a form of structural operational semantics. We will not consider jump instructions here, so need only consider one instruction at a time.[3]

The operational semantics is given in terms of configurations of abstract syntax and labelled arrows. We regard configurations themselves as a form of instruction; in effect, a closure. Any changes to the heap or environment are given implicitly in the arrow rather than being explicitly part of the configuration. This affords some modularity for then the semantics of those instructions which do not use the heap or environment can be given as such and then extended verbatim.

In Section 4, we define abstract types which are common to the two formats. It is this type structure which is used to define the logical relations. In Section 5 we give an operational semantics which is independent of the underlying class/CAP file format. The structure of the class/CAP file need not be visible to the operational semantics. We need only be able to extract certain data corresponding to a particular method, such as the appropriate constant pool, so define auxiliary functions for accessing such data.

In Sections 6 and 7, we give the specific details of the name and token formats, respectively, defined as interpretations of types and auxiliary functions, $[\![.]\!]_{name}$ and $[\![.]\!]_{tok}$.

In Section 8, we define the logical relation, $\{R_\theta\}_{\theta \in Abstract\_type}$. It is convenient to informally group the definition into several levels. First of all, there are various basic types (byte, short, *etc.*), γ, for which we have $R_\gamma = id_\gamma$. Then there are the references, ι, such as package and class references, for which the relation R_ι represents the tokenisation of named items.

The constraints on the componentisation are expressed in R_κ, where κ includes method information structures, constant pools, and so on. This represents the relationship between components in CAP files and the corresponding entities in class files.

Using the above three families of relations we can define R_θ for each type, θ, where

$$\theta ::= \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \rightarrow \theta' \mid \theta + \theta' \mid \theta^*$$

The family of relations, $\{R_\theta\}_{\theta \in Type}$, represents the overall construction of components in the CAP file format from a class file. The relations are 'logical' in sense that the definitions for defined types follows automatically. For example, we define the type of the environment and heap as

$$\texttt{Environment} = \texttt{Package\_ref} \rightarrow \texttt{Package}$$

$$\texttt{Heap} = \texttt{Object\_ref} \rightarrow \texttt{Object}$$

and so the definition of $R_{\texttt{Environment}}$ follows from those of $R_{\texttt{Package\_ref}}$, $R_{\texttt{Package}}$ and the (standard[4]) construction of $R_{\_ \rightarrow \_}$; similarly for $R_{\texttt{Heap}}$. In Section 9, we use this semantic format to prove the correctness. The proof has two parts:

1. We prove that all auxiliary functions are representation independent; that is, if $f : \theta \rightarrow \theta'$ then we have $[\![f]\!]_{name} \; R_{\theta \rightarrow \theta'} \; [\![f]\!]_{tok}$.

2. Then, it is straightforward to prove that all instructions are representation independent, using part 1. It is convenient to view the operational semantics of bytecode as giving an interpretation

$$[\![code]\!] : \texttt{State} \rightarrow \texttt{Bytecode} \times \texttt{State}$$

   where

$$\texttt{State} = \texttt{Global\_state} \times \texttt{Local\_state}$$

$$\texttt{Global\_state} = \texttt{Environment} \times \texttt{Heap}$$

$$\texttt{Local\_state} = \texttt{Operand\_stack} \times \texttt{Local\_variables} \times \texttt{Class\_ref}$$

We can conclude, therefore, that if a transformation satisfies certain constraints (formally expressed by saying that it is contained in R) then it is correct, in the sense that no difference can be observed in the two semantics.

Finally, we make some concluding remarks in Section 10.

[3]Actually, jumps should not present a problem. We could add labels to the abstract syntax, and keep the code as a component of the global state. An auxiliary function would search for the label and evaluation would proceed from that point onwards.

[4]It is simpler to allow functions to be partial.

4 Abstract Types

We use types to structure the transformation. These are not the types of the Java Card language, but rather are based on the simply-typed lambda calculus with sums, products and lists. We use record types with the actual types of fields (drawn from the official specification where not too confusing) serving as labels. We write elements of sum types in the form $\langle tag, value \rangle$. Occasionally we use terms as singleton types, such as `0xFFFF` and `0`. We use a set-theoretic notation where convenient.

There are two sorts of types: *abstract* and *concrete*. The idea is that abstract types are those we can think of independently of a particular format. The concrete types are the particular realisations of these, as well as types which only make sense in one particular model. For example, `CP_index` is the abstract type of indices into a constant pool for a given package. In the name interpretation, this is modelled by a class name and an index into the constant pool of the corresponding class file, *i.e.* $\texttt{Class\_name} \times \texttt{Index}$ where `Index` is a concrete type. In the token format, however, since all the constant pools are merged, we have $[\![\texttt{CP\_index}]\!]_{tok} = \texttt{Package\_tok} \times \texttt{Index}$. Another example is the various distinctions that are made between method and field references in CAP files, but not class files, and which are not relevant at the level of the operational semantics, which concerns terms of abstract types.

We arrange the types so that as much as possible is common between the two formats. For example, it is more convenient to uniformly define environments as mappings of type $\texttt{Package\_ref} \rightarrow \texttt{Package}$, with `Package` interpreted as $\texttt{Class\_name} \rightarrow \texttt{Class\_file}$ or `CAP_file`.

There is a 'type of types' for the two forms of data type in Java Card — primitive types, the simple types supported directly on the card, and reference types.

$$\texttt{Type} = \{\texttt{Boolean}, \texttt{Byte}, \texttt{Short}\} + \texttt{Reference\_type}$$

We have not included `Int`, which is optional.

$$\texttt{Reference\_type} = \texttt{Array\_type} + \texttt{Class\_ref}$$

We use a separate type, `Object_ref`, to refer to objects on the heap. The objects themselves contain a reference to the appropriate class or array of which they form an instance.

The type `Word` is a platform specific abstract unit of storage. All we need know is that object references and the basic types, `Byte`, `Short` and `Boolean`, can be stored in a `Word`. Rather than use an explicit coercion, we assume

$$\texttt{Word} = \texttt{Object\_ref} + \texttt{Null} + \texttt{Boolean} + \texttt{Byte} + \texttt{Short}$$

Thus a word is (*i.e.* represents) either a reference (possibly null) or an element of a primitive type. Furthermore, we define

$$\texttt{Value} = \texttt{Word}$$

Although this is not strictly necessary, there is a conceptual distinction. If we were to introduce values of type `Int`, then a value could be either a word or a double word.

There are several forms of reference[5]:

$$\texttt{Package\_ref} \mid \texttt{Class\_ref} \mid \texttt{Field\_ref} \mid \texttt{Method\_ref} \mid \texttt{Interface\_method\_ref}$$

We distinguish `Package` from `Package_ref`, and similarly for the other items. Note that a *reference* is a composite entity which can be context dependent (eg. a class reference can be in internal or external forms). We assume, however, that sufficient information is given so that references make sense globally. For example, class names are fully qualified, and class tokens are paired with a package token. We take field and method references to be to particular members of some class, and so contain a class reference. By contrast, an *identifier* is a name or a token. We do not use identifiers at the abstract level though. There is not a specific form of reference for interfaces. These are taken to be a particular form of class reference.

$$\texttt{CP\_info} = \texttt{Class\_ref} + \texttt{Method\_ref} + \texttt{Field\_ref}$$

Since, in Java, only constants of 'big' type can appear in the constant pool, these are not present in Java Card. Moreover, since we consider the constant pool to be fully resolved, we do not need separate entries for names.

[5] Which we distinguish from `Reference_type`

The index type, `CP_index`, expresses the rearrangement of the constant pools. We introduce a separate index type, `SM_index`, for the method references used by the `invokespecial` instruction. These are treated differently since the corresponding entries are replaced with supermethod references during conversion.

There is a type for method information structures

$$\texttt{Method\_info}$$

Since types are coded differently in the two formats we introduce an abstract type

$$\texttt{Type\_code}$$

We follow Bertelsen [Ber97] in considering the constant pool to be partially resolved. For example, rather than taking a method reference in the class file format to be a tag (which can be ignored) plus a class index (the reference at which, in turn, contains an index to a string constant) and a signature (that is, a name-and-type index, which also contains indices), we just represent it as a tuple of class name and descriptor.

Finally, we use types for grouping parts of the transformation.

$$\texttt{Package} \mid \texttt{Class} \mid \texttt{Pack\_methods} \mid \texttt{Pack\_fields} \mid \texttt{Constant\_pool}$$

Using these basic types, we can then construct complex types using the usual type constructors. We will use (non-dependent) sum, product, function and list types (denoted θ^*). We do not use lists much since it is simpler to represent tables as functions with a domain of indices rather than lists.

The virtual machines are formalised as follows:

$$\texttt{Locals} = \texttt{Nat} \rightarrow \texttt{Word}$$

$$\texttt{Config} = \texttt{Bytecode} \times \texttt{Word}^* \times \texttt{Locals} \times \texttt{Class\_ref}$$

$$\texttt{Bytecode} = \texttt{Instruction} + (\texttt{Bytecode} \times \texttt{Bytecode}) + \texttt{Config}$$

$$\texttt{Instruction} = \texttt{Nop} + \texttt{Invokevirtual}\,\texttt{CP\_index} + \texttt{Checkcast}\,\texttt{Typecode} + \texttt{Invokespecial}\,\texttt{SM\_index} \cdots$$

We treat the instructions as types, and give full details in the next section. Since bytecode contains indices into the constant pool, it changes during tokenisation. We could model this in a number of ways. One possibility would be to treat the indices themselves as auxiliary functions (Börger and Schulte [BS98] do something similar to this), although this is essentially equivalent to the approach we have taken. The essential point is that `Bytecode` depends on `CP_index` and, indeed, other types.

The tokenisation requires us to make various distinctions, such as between static and instance fields, which are not needed for some auxiliary functions.

There are other constructed types, which we need not be directly concerned with. For example:

$$\texttt{Class\_inst\_obj} = \texttt{Class\_ref} \times \texttt{IV}$$

In Java, array types contain the array dimension, whereas array objects contain the length. Since arrays are unidimensional in Java Card, the dimension is unnecessary.

$$\texttt{Array\_obj} = \texttt{Nat} \times \texttt{Array\_type} \times (\texttt{Nat} \rightarrow \texttt{Value})$$

$$\texttt{Array\_type} = \texttt{Primitive\_type} + \texttt{Class\_ref}$$

$$\texttt{Object} = \texttt{Class\_inst\_obj} + \texttt{Array\_obj}$$

$$\texttt{IV} = \texttt{Field\_ref} \rightarrow \texttt{Value}$$

We will assume that the classes are grouped correctly according to their package.

$$\texttt{Environment} = \texttt{Package\_ref} \rightarrow \texttt{Package}$$

$$\texttt{Heap} = \texttt{Object\_ref} \rightarrow \texttt{Object}$$

5 Operational Semantics

The official specification of the JCVM (and JVM) is given in terms of *frames*. This is the state of the current method invocation, together with any other useful data. There is some choice for how to model frames and the various formalisations of bytecode semantics in the literature differ slightly in their approach. Although the official specification also mentions a reference to the current constant pool we will calculate this from the current class reference.

We abstract away from details of program counters and (literal) byte codes, and instead formalise the code as abstract syntax. We introduce the notion of *configuration*, consisting of the (abstract syntax of the) code of the current method still to be executed, the operand stack, the local variables, and the current class reference. We model local variables as a partial function, but represent this as a list.

To account for method invocations, it is convenient to allow a configuration itself to be considered as an instruction. When a method is invoked, the next instruction becomes a current configuration. Instead of a stack of frames, then, we have a single piece of 'code' (in this general sense). This form of closure is equivalent to the traditional idea of a call stack.

We use a single-step SOS, Since execution does not terminate, as such, we introduce an artificial instruction `nop` to signify the termination of an instruction. The following two rules are standard for SOS:

$$\frac{\langle b_1, ops, l, c\rangle \Rightarrow \langle b_1', ops', l', c'\rangle}{\langle b_1; b_2, ops, l, c\rangle \Rightarrow \langle b_1'; b_2, ops', l', c'\rangle} \qquad \frac{\langle b_1, ops, l, c\rangle \Rightarrow \langle \mathtt{nop}, ops', l', c'\rangle}{\langle b_1; b_2, ops, l, c\rangle \Rightarrow \langle b_2, ops', l', c'\rangle}$$

For configurations we use the rule:

$$\frac{f \Rightarrow f'}{\langle \mathtt{Config}\ f, ops, l, c\rangle \Rightarrow \langle \mathtt{Config}\ f', ops, l, c\rangle}$$

We will write $\mathtt{Config}\,(b, o, l, c)$ as $\langle b, o, l, c\rangle$.

The method invocation instructions (and others) take an argument which is an index into either the constant pool of a class file, or into the constant pool component of a CAP file. This means that the 'concrete' bytecode is itself dependent on the implementation.

Thus we define a transition relation

$$\Rightarrow\ \subseteq\ \mathtt{Config} \times \mathtt{Arrow} \times \mathtt{Config}$$

where

$$\mathtt{Config} = \mathtt{Bytecode} \times \mathtt{Word}^* \times \mathtt{Locals} \times \mathtt{Class\_ref}$$

$$\mathtt{Arrow} = \mathtt{Global\_state} \to \mathtt{Global\_state}$$

$$\mathtt{Global\_state} = \mathtt{Environment} \times \mathtt{Heap}$$

$$\mathtt{Bytecode} = \mathtt{Instruction} + (\mathtt{Bytecode} \times \mathtt{Bytecode}) + \mathtt{Config}$$

$$\begin{aligned}\mathtt{Instruction} \ = \ & \mathtt{Nop} + \mathtt{Invokevirtual}\ \mathtt{CP\_index} + \mathtt{Invokestatic}\ \mathtt{CP\_index} + \\ & \mathtt{Invokeinterface}\ \mathtt{CP\_index} + \mathtt{Invokespecial}\ \mathtt{SM\_index} + \mathtt{Return} + \\ & \mathtt{New}\ \mathtt{CP\_index} + \mathtt{Putstatic}\ \mathtt{CP\_index} + \mathtt{Getstatic}\ \mathtt{CP\_index} + \\ & \mathtt{Putfield}\ \mathtt{CP\_index} + \mathtt{Getfield}\ \mathtt{CP\_index} + \mathtt{Checkcast}\ \mathtt{Typecode} + \\ & \mathtt{Instanceof}\ \mathtt{Typecode}\end{aligned}$$

The rules are given in the following form.

$$\frac{\begin{array}{c}\text{hypothesis 1}\\ \vdots \\ \text{hypothesis } n\end{array}}{\mathit{config} \overset{\mathit{statechange}}{\Longrightarrow} \mathit{config}}$$

Hypotheses are either conditions or assignments. In fact, almost all hypotheses here are assignments. The only conditions used are in `invokespecial`, where the predicate `super_invocation` is used to choose between two behaviours, and in `checkcast`, where an exception can be raised. We make liberal use of wildcards '_' in assignments to suppress unimportant details.

When the heap and environment do not change, we will not bother to write the label explicitly. Since at most one transition can be a hypothesis, we adopt the convention that, unless stated otherwise, the label on the arrow of the conclusion is the same as that on this hypothesis. Moreover, the arrow on a transition with no transitions for hypotheses is the identity, $id_{(heap,env)}$. We can implicitly use *heap* and *env* in the hypotheses.

If an instruction changes the heap or the environment, then we label the arrow with an operation which abstracts the effect on the global state. For example, $add(r, o)$ is the arrow

$$\langle h, e\rangle \mapsto \langle h + (r \mapsto o), e\rangle$$

We factor out those tedious parts of the semantics which are common to most instructions into a number of auxiliary selector functions. We adopt the convention of using capitalised names for types, and lower case names for the corresponding selector functions. So, we have:

$$\texttt{constant\_pool} : \texttt{Class\_ref} \to \texttt{Constant\_pool}$$

where `Constant_pool` is the abstract type of the constant pool.

We give the semantics of those instructions which make use of the constant pool and class hierarchy, namely[6]:

- the various method invocation instructions,
- instructions for instantiating and accessing classes, and
- the two type checking instructions.

By virtue of the verification phase, we can assume that various static checks have been carried out. We use a number of auxiliary functions. Certain functions have preconditions, which we take as concomitant with the well-formedness of the class file. Some functions use the environment and the heap. Rather than pass these as explicit arguments we will assume them to be globally accessible.

The `lookup` function takes the class reference where a method is declared, together with the actual method reference (which contains the actual class reference), and returns the class reference where the method is defined together with the code. We assume that the declared and actual class are in the same class hierarchy.

$$\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \to \texttt{Class\_ref} \times \texttt{Bytecode}$$

We use a separate function `lookup_int` for looking up interface methods, since they are treated differently.

$$\texttt{lookup\_int} : \texttt{Class\_ref} \times \texttt{Interface\_method\_ref} \to \texttt{Class\_ref} \times \texttt{Bytecode}$$

`instance_fields`(c) returns the default values for the instance fields in c.

$$\texttt{instance\_fields} : \texttt{Class\_ref} \to (\texttt{Field\_ref} \to \texttt{Value})$$

`static_val`(f) returns the static value in field f.

$$\texttt{static\_val} : \texttt{Field\_ref} \to \texttt{Value}$$

`instOf`(r, o) returns a boolean corresponding to whether the object o can be case to reference type r.

$$\texttt{instOf} : \texttt{Reference\_type} \times \texttt{Object} \to \texttt{Bool}$$

The functions *add* and `update` are used in formalising the operational semantics. $add(r, o)$ puts the new binding $r \mapsto o$ on the heap, and `update`(f, v) changes the binding of static field f to value v. We can define $add : \texttt{Object\_ref} \times \texttt{Object} \to \texttt{Statechange}$ independently of the underlying format, so do not regard it as an auxiliary function. In constrast, since we assume that the values of static fields are stored in the class/CAP file (though this is not clear), the `update` function changes the environment and so depends on the format.

$$\texttt{update} : \texttt{Field\_ref} \times \texttt{Value} \to \texttt{Statechange}$$

where

$$\texttt{Statechange} = \texttt{Global\_state} \to \texttt{Global\_state}$$

[6]With the exception of `aastore`. We also ignore `invokevirtual` on array objects, and native methods.

$$\texttt{Global\_state} = \texttt{Environment} \times \texttt{Heap}$$

The structure of the class/CAP file need not be visible to the operational semantics. We need only be able to extract certain data corresponding to a particular method or class, such as the appropriate constant pool.

`super`(c) returns a reference to the superclass of c.

$$\texttt{super} : \texttt{Class\_ref} \rightarrow \texttt{Class\_ref}$$

The function `super_invocation` is used in the semantics of the `invokespecial` instruction, and returns a boolean corresponding to whether or not an invocation should be of a supermethod.

$$\texttt{super\_invocation} : \texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Bool}$$

The functions `method_class`, `method_code` and `method_nargs` return the class, code and number of arguments for a given method reference.

$$\texttt{method\_class} : \texttt{Method\_ref} \rightarrow \texttt{Class\_ref}$$

We only use `method_class` for static (and super) method references.

$$\texttt{method\_code} : \texttt{Method\_ref} \rightarrow \texttt{Bytecode}$$

$$\texttt{method\_nargs} : \texttt{Method\_ref} \rightarrow \texttt{Nat}$$

The function `method_code` assumes that the appropriate method is defined at the given class and does not do any searching. It is used in the semantics of the `invokestatic` and `invokespecial` instructions.

Finally, since reference types are represented differently in the two formats, we use a function `reference_type` to abstract away from the concrete format (expressed using a natural and/or an index into the constant pool) and return the actual type:

$$\texttt{reference\_type} : \texttt{Type\_code} \times \texttt{Class\_ref} \rightarrow \texttt{Reference\_type}$$

5.1 Method Invocation

Note that whereas some of these operations have class loading aspects in Java, Java Card does not have dynamic class loading.

5.1.1 Invokevirtual

The procedure is:

1. The two byte index, i, into the constant pool is resolved to get the declared method reference containing the declared class reference and a method identifier (either a signature or token).
2. The number of arguments to the method is calculated.
3. The object reference, r, is popped off the operand stack.
4. Using the heap, we get $heap(r) = \langle act\_cref, \_\rangle$, the actual class reference (fully qualified name or a package/class token pair).
5. We then do $\texttt{lookup}(act\_cref, dec\_mref)$, getting the class where the method is implemented, and its bytecode. The lookup function is used with respect to the class hierarchy (environment).
6. A new configuration is created for this method and evaluation proceeds from there.

$$\begin{array}{ll}
dec_mref := \texttt{constant_pool}\ (c)(i) & \text{get declared method reference from constant pool} \\
n := \texttt{method_nargs}(stat_mref) & \text{get number of arguments} \\
\langle act_cref, _\rangle := heap(r) & \text{get actual class reference from heap} \\
\langle m_class, m_code\rangle := \texttt{lookup}(act_cref, dec_mref) & \text{look up method} \\
\hline
\multicolumn{2}{l}{\langle \texttt{invokevirtual}\ i, a_1 \ldots a_n :: r :: s, l, c\rangle \Rightarrow \langle\langle m_code, \langle\rangle, a_1 \ldots a_n :: r, m_class\rangle, s, l, c\rangle}
\end{array}$$

5.1.2 Invokestatic

Invocation of class (static) methods.

$$\frac{\begin{array}{ll} dec\_mref := \texttt{constant\_pool}\ (c)(i) & \text{get declared method reference from constant pool} \\ n := \texttt{method\_nargs}(stat\_mref) & \text{get number of arguments} \\ m\_class := \texttt{method\_class}\ (dec\_mref) & \text{get class of method} \\ m\_code := \texttt{method\_code}\ (dec\_mref) & \text{get method code} \end{array}}{\langle \texttt{invokestatic}\ i, a_1, \ldots, a_n :: s, l, c\rangle \Rightarrow \langle\langle m\_code, \langle\rangle, a_1, \ldots, a_n, m\_class\rangle, s, l, c\rangle}$$

5.1.3 Invokeinterface

By virtue of the verification phase, we can assume that various static checks have been carried out and that, for example, the resolved method is not an initialization method (`<init>` or `<clinit>`), that *act_cref* implements the interface, and that there are $n-1$ arguments on the operand stack. According to [LY97], the n is a historical redundancy. A run-time exception can be thrown if the object reference r is null.

$$\frac{\begin{array}{ll} int\_mref := \texttt{constant\_pool}\ (c)(i) & \text{get declared method reference in constant pool} \\ n := \texttt{method\_nargs}(int\_mref) & \text{get number of arguments} \\ \langle act\_cref, \_\rangle := heap(r) & \text{get class name from heap} \\ \langle m\_class, m\_code\rangle := \texttt{lookup\_int}(act\_cref, int\_mref) & \text{look up interface method} \end{array}}{\langle \texttt{invokeinterface}\ i, a_1, \ldots, a_n :: r :: s, l, c\rangle \Rightarrow \langle\langle m\_code, \langle\rangle, a_1, \ldots, a_n, m\_class\rangle, s, l, c\rangle}$$

5.1.4 Invokespecial

The `invokespecial` instruction has two behaviours, depending on the modifiers of the actual class, and the kind of method invoked. Either the superclass of the actual class is searched, or a method of the actual class itself is used, which is allowed to be private or an initialization method. The reasoning is described in [LY97] as:

$$\begin{array}{l} \texttt{if}\ \neg\texttt{<init>} \wedge \neg\texttt{private} \wedge (actual\_class < method\_class) \wedge \texttt{super}(actual) \\ \texttt{then} \\ \quad super\_method \\ \texttt{else if}\ \ \texttt{<init>}\ \wedge uninitialised(object\_ref) \\ \texttt{then} \\ \quad init\_method \\ \texttt{else}\ /\!*\ \text{may be private}\ *\!/ \\ \quad actual\_class\_method \end{array}$$

An analysis is carried out to determine which of these cases holds. Although this might be carried out at run-time in the class format, in the CAP format it is analysed statically as part of the conversion process, so that the method reference indicates explicitly whether or not it is a super invocation. However, we abstract this test out into a function `super_invocation`.

Hence it is unnecessary to get the class name from the heap at runtime. A static dataflow analysis during verification ensures that methods are always initialized before use.

For `invokespecial` i, on a 'superclass' instance method. Note that we call the lookup with the superclass of the actual class:

$$\frac{\begin{array}{ll} dec\_mref := \texttt{constant\_pool}\ (c)(i) & \text{get declared method reference in constant pool} \\ n := \texttt{method\_nargs}(dec\_mref) & \text{get number of arguments} \\ \texttt{super\_invocation}\ (c, dec\_mref) & \text{check for super invocation} \\ super\_cref := \texttt{super}(\texttt{method\_class}(dec\_mref)) & \text{get superclass of method call} \\ \langle m\_cref, m\_code\rangle := \texttt{lookup}(super\_cref, dec\_mref) & \text{look up method from superclass} \end{array}}{\langle \texttt{invokespecial}\ i, a_1, \ldots, a_n :: r :: s, l, c\rangle \Rightarrow \langle\langle m\_code, \langle\rangle, a_1, \ldots, a_n, m\_cref\rangle, s, l, c\rangle}$$

For `invokespecial` i on instance initialization methods, and methods of the actual class (which may be private), there is no lookup:

$$\begin{array}{ll}
dec_mref := \texttt{constant_pool}\ (c)(i) & \text{get declared method reference in constant pool} \\
n := \texttt{method_nargs}(dec_mref) & \text{get number of arguments} \\
\neg\texttt{super_invocation}\ (c, dec_mref) & \text{check not super invocation} \\
m_class := \texttt{method_class}(dec_mref) & \text{get class of method} \\
m_code := \texttt{method_code}(dec_mref) & \text{get method code} \\
\hline
\langle \texttt{invokespecial}\ i, a_1, \ldots, a_n :: s, l, c \rangle \Rightarrow \langle \langle m_code, \langle\rangle, a_1, \ldots, a_n, m_class \rangle, s, l, c \rangle &
\end{array}$$

5.1.5 Method Return

The method return instructions do not use the configuration of the current method.

$$\frac{}{\langle \langle \texttt{return}, \_, \_, \_ \rangle, ops, l, c \rangle \Rightarrow \langle \texttt{nop}, ops, l, c \rangle}$$

5.2 Class Instructions

As mentioned above, Java Card does not support dynamic class loading so these operations all assume that the appropriate classes are loaded.

5.2.1 Class Instantiation

We can assume that the class is not abstract or an interface, that the class is accessible, and that the operand stack will not overflow.

The function `instance_fields` returns a mapping which gives the default field values of a class:

$$\texttt{instance\_fields} : \texttt{Class\_ref} \to (\texttt{Field\_ref} \to \texttt{Value})$$

$$\texttt{static\_val} : \texttt{Field\_ref} \to \texttt{Value}$$

Recall that we define $add(r, o)$, which adds the binding of reference r to object o to the heap, as:

$$add(r, o) = \lambda \langle e, h \rangle \ . \ \langle e, h + \{r \mapsto o\} \rangle$$

Since we can define add independently of format we do not regard it as an auxiliary function.

$$\begin{array}{ll}
c_ref := \texttt{constant_pool}\ (c)(i) & \text{get class reference} \\
iv := \texttt{instance_fields}(c_ref) & \text{compute default values} \\
o := \langle c_ref, iv \rangle & \text{construct new object} \\
r \in \texttt{Object_ref} \backslash dom(heap) & \text{find new reference} \\
\hline
\langle \texttt{new}\ i, ops, l, c \rangle \stackrel{add(r,o)}{\Longrightarrow} \langle \texttt{nop}, r :: ops, l, c \rangle &
\end{array}$$

We assume the existence of a deterministic choice function for selecting a new object reference. We do not define an auxiliary function, however, since we have not given a concrete implementation of the type `Object_ref` in either model. This makes the proof of equivalence slightly easier though, strictly speaking, all that is necessary is that the transition rules are *observably* deterministic.

5.2.2 Class Fields

We use the function, `update`, which takes a class, a static field of that class, and a value of compatible type, and overlays the change to the environment given by updating the field with that value. It is because of this that the environment can change.

$$\begin{array}{ll}
f_ref := \texttt{constant_pool}\ (c)(i) & \text{get field reference} \\
\hline
\langle \texttt{putstatic}\ i, v :: ops, l, c \rangle \stackrel{\texttt{update}(f_ref, v)}{\Longrightarrow} \langle \texttt{nop}, ops, l, c \rangle &
\end{array}$$

$$\begin{array}{ll}
f_ref := \texttt{constant_pool}\ (c)(i) & \text{get field reference} \\
v := \texttt{static_val}(f_ref) & \text{get static value} \\
\hline
\langle \texttt{getstatic}\ i, ops, l, c \rangle \Rightarrow \langle \texttt{nop}, v :: ops, l, c \rangle &
\end{array}$$

5.2.3 Instance Fields

The intructions for putting and getting the instance field of an object, r, where the reference r is on the heap, assume that r is initialized.

$$\frac{\begin{array}{ll} f\_ref := \texttt{constant\_pool}\ (c)(i) & \text{get field reference} \\ \langle c'\_ref, iv\rangle := heap(r) & \text{get object from heap} \\ o := \langle c'\_ref, iv + \{f\_ref \mapsto v\}\rangle & \text{update object} \end{array}}{\langle \texttt{putfield}\ i, v :: r :: ops, l, c\rangle \stackrel{add(r,o)}{\Longrightarrow} \langle \texttt{nop}, ops, l, c\rangle}$$

$$\frac{\begin{array}{ll} f\_ref := \texttt{constant\_pool}\ (c)(i) & \text{get field reference} \\ \langle c'\_ref, iv\rangle := heap(r) & \text{get object from heap} \\ v := iv(f\_ref) & \text{get field value} \end{array}}{\langle \texttt{getfield}\ i, r :: ops, l, c\rangle \Rightarrow \langle \texttt{nop}, v :: ops, l, c\rangle}$$

5.3 Type Checking

The $\texttt{instOf} : \texttt{Reference\_type} \times \texttt{Object} \to \texttt{Bool}$ predicate formalises when an object can be cast to a type. The object reference, r, is left on the stack. If the condition fails, the instruction throws a `CheckCastException`. We do not model this.

$$\frac{\begin{array}{ll} d := \texttt{reference\_type}(tc, c) & \text{get type descriptor} \\ r \neq null \Rightarrow r \in dom(heap) \wedge \texttt{instOf}(d, heap(r)) & \text{check cast is valid} \end{array}}{\langle \texttt{checkcast}\ tc, r :: s, l, c\rangle \Rightarrow \langle \texttt{nop}, r :: s, l, c\rangle}$$

The `instanceof` instruction has similar semantics to `checkcast`, though does not throw an exception. Instead, the object reference on the operand stack is replaced with a bit representing the result of the check.

$$\frac{\begin{array}{ll} d := \texttt{reference\_type}\ (tc, c) & \text{get type descriptor} \\ b := \texttt{if}\ \texttt{instOf}(d, heap(r))\ \texttt{then}\ 1\ \texttt{else}\ 0 & \text{compute `instance bit'} \end{array}}{\langle \texttt{instanceof}\ tc, r :: s, l, c\rangle \Rightarrow \langle \texttt{nop}, b :: s, l, c\rangle}$$

6 Name Interpretation

Classes are described by fully qualified names, whereas methods and fields are given signatures, consisting of an unqualified name and a type, together with the class of definition. We assume a function $pack\_name$ which gives the package name of a class name.

The data is arranged into class files, each of which contains all the information corresponding to a particular class. We only give the detail of those parts used here. We group the class files by package into a global environment. Thus $env\_name(p)(c)$ denotes the class file in package p with name c.

6.1 Types

$$[\![\texttt{Package}]\!]_{name} = \texttt{Class\_name} \to \texttt{Class\_file}$$

$$[\![\texttt{Package\_ref}]\!]_{name} = \texttt{Package\_name}$$

$$[\![\texttt{Class\_ref}]\!]_{name} = \texttt{Class\_name}$$

$$[\![\texttt{Field\_ref}]\!]_{name} = \texttt{Class\_name} \times \texttt{Field\_name}$$

$$[\![\texttt{Method\_ref}]\!]_{name} = \texttt{Class\_name} \times \texttt{Sig}$$

$$\texttt{Sig} = \texttt{Method\_name} \times [\![\texttt{Type}]\!]^{*}_{name}$$

$$[\![\texttt{Interface\_method\_ref}]\!]_{name} = \texttt{Class\_name} \times \texttt{Sig}$$

$$[\![\texttt{Constant\_pool}]\!]_{name} = [\![\texttt{CP\_index}]\!]_{name} \to [\![\texttt{CP\_info}]\!]_{name}$$

$$[\![\texttt{CP\_index}]\!]_{name} = \texttt{Class\_name} \times \texttt{Index}$$

$$[\![\texttt{Class}]\!]_{name} = \texttt{Class\_file}$$

$$\texttt{Class\_file} = \texttt{Class\_flags} \times (\texttt{Class\_name} + \texttt{Void}) \times \texttt{Fields\_item} \times \texttt{Methods\_item} \times \texttt{Constant\_pool\_item} \times \texttt{Class\_name}$$

$$\texttt{Class\_flags} = \texttt{Class\_flag} \rightarrow \texttt{Bool}$$

$$\texttt{Class\_flag} = \texttt{Public} + \texttt{Final} + \texttt{Super} + \texttt{Interface} + \texttt{Abstract}$$

Since there is no overloading of fields, a field signature is just a name.

$$\texttt{Fields\_item} = \texttt{Field\_name} \rightarrow \texttt{Field\_info}$$

$$\texttt{Field\_info} = \texttt{Field\_flags} \times [\![\texttt{Type}]\!]_{name} \times \texttt{Value}$$

The `Value` field is only required for static fields.

$$\texttt{Field\_flags} = \texttt{Field\_flag} \rightarrow \texttt{Bool}$$

$$\texttt{Field\_flag} = \texttt{Public} + \texttt{Private} + \texttt{Protected} + \texttt{Static} + \texttt{Final}$$

$$[\![\texttt{Pack\_fields}]\!]_{name} = \texttt{Class\_name} \rightarrow \texttt{Fields\_item}$$

$$[\![\texttt{Pack\_methods}]\!]_{name} = \texttt{Class\_name} \rightarrow \texttt{Methods\_item}$$

$$\texttt{Methods\_item} = \texttt{Sig} \rightarrow \texttt{Method\_info}$$

$\texttt{Method\_info} =$
$\texttt{Method\_flags} \times \texttt{Sig} \times ([\![\texttt{Type}]\!]_{name} + \texttt{Void}) \times \texttt{Exception\_classes} \times \texttt{Maxstack} \times \texttt{Maxlocals} \times \texttt{Bytecode} \times$
$\texttt{Exception\_handlers}$

The signature is not considered to include the return type. We assume that this signature is the same as the argument given to the methods item. We represent the flags as a predicate (boolean function) over the possible settings. There are various constraints on the flags which we do not consider here.

$$\texttt{Method\_flags} = \texttt{Method\_flag} \rightarrow \texttt{Bool}$$

$$\texttt{Method\_flag} = \texttt{Public} + \texttt{Private} + \texttt{Protected} + \texttt{Static} + \texttt{Final} + \texttt{Native} + \texttt{Abstract}$$

$$\texttt{Constant\_pool\_item} = \texttt{Index} \rightarrow \texttt{CP\_info}$$

Finally,

$$[\![\texttt{Type\_code}]\!]_{name} = [\![\texttt{CP\_index}]\!]_{tok}$$

We do not need to give interpretations for the constructed abstract types. It follows, for example, that

$$[\![\texttt{CP\_info}]\!]_{name} = [\![\texttt{Class\_ref}]\!]_{\texttt{name}} + [\![\texttt{Field\_ref}]\!]_{\texttt{name}} + [\![\texttt{Method\_ref}]\!]_{\texttt{name}} + [\![\texttt{Interface\_method\_ref}]\!]_{\texttt{name}}$$

6.2 Auxiliary Functions

We use an extended lambda calculus to define the functions. In addition to typed abstractions and pairs, we use conditionals, case expressions, let expressions, function overloading, and pattern matching in both lets and abstractions. We also use set-theoretic constructs, such as union and the map notation.

$(\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Class\_ref} \times \texttt{Bytecode})$

$[\![\texttt{lookup}]\!]_{name} = \texttt{lookup\_name}$

There are a number of possibilities for how method lookup should be defined, depending on the definition of inheritance. For example, [Ber97, Pus98] use a 'naive' lookup which does not take account of visibility modifiers. A fuller discussion of this appears in [Seg99].

```
lookup_name (act_class, (sig, dec_class)) =
              let
               dec_pk  = pack_name(dec_class)
               act_pk  = pack_name(act_class)
               (_,_,_,_,_,meth_dec,_,_)
                      = env_name (dec_pk) (dec_class)
               (_,super,_,_,_,meth_act,_,_)
                      = env_name (act_pk) (act_class)
               (dec_flags,_,_,_,_,_,_) = meth_dec(sig)
               in
               if meth_act(sig) = undefined
               then lookup_name(super, (sig, dec_class))
               else if
                    dec_flags(protected) or dec_flags(public)
                    or act_pk = act_pk
                    then let (_,_,_,_,_,code,_) = meth_act(sig)
                            in (act_class,code)
                    else  lookup_name(super, (sig, dec_class))
```

$(\texttt{lookup\_int} : \texttt{Class\_ref} \times \texttt{Interface\_method\_ref} \to \texttt{Class\_ref} \times \texttt{Bytecode})$

$[\![\texttt{lookup\_int}]\!]_{name} = \texttt{lookup\_name}$

$(\texttt{static\_val} : \texttt{Field\_ref} \to \texttt{Value})$

$\lambda\langle c, \langle f, t\rangle\rangle : \texttt{Field\_ref} \,.\, env\_name(pack\_name(c))(c).Fields\_item(f)$

$(\texttt{update} : \texttt{Field\_ref} \times \texttt{Value} \to \texttt{Statechange})$

$$\begin{array}{l}
\texttt{update}\,((c_name, f_name, _), v) = \lambda\langle e, h\rangle : \texttt{Environment} \times \texttt{Heap}\,. \\
\quad \langle \lambda p' : \texttt{Package_name}\,.\,\lambda cn : \texttt{Class_name}\,.\,\texttt{if}\ cn = c_name\ \texttt{then} \\
\qquad \texttt{let}\ \langle f, s, fields, m, cp, n\rangle = e(p')(c_name) \\
\qquad \langle flags, d, value\rangle = fields(f_name)\ \texttt{in} \\
\qquad \langle f, s, fields + \{f_name \mapsto \langle flags, d, v\rangle\}, m, cp, n\rangle \\
\qquad \texttt{else}\ e(p')(cn)\ , \\
\qquad\quad h\rangle
\end{array}$$

Because of our assumption on the well-formedness of environments, $p' = pack\_name(cn)$.

$(\texttt{instance\_fields} : \texttt{Class\_ref} \to (\texttt{Field\_ref} \to \texttt{Value}))$

$$\begin{array}{l}
\texttt{instance_fields}(c_name) = \\
\quad \texttt{let}\ \langle _, super, _, fields, _, _, _, _\rangle = env_name(pack_name(c_name))(c_name)\ \texttt{in} \\
\quad\ \texttt{if}\ c_name = \texttt{javacard.lang.Object}\ \texttt{then}\ \texttt{instance_fields1}(c_name, fields) \\
\quad\ \texttt{else}\ \texttt{instance_fields}(super) \cup \texttt{instance_fields1}(c_name, fields)
\end{array}$$

The function *default* computes the default values for each type.

$$\begin{array}{l}
\texttt{instance_fields1}(c_name, fields) = \\
\quad \{\ (\ \langle c_name, f_name\rangle \mapsto default(d)\ )\ \mid\ fields(f_name) = \langle flags, d, _\rangle\ \wedge\ \neg flags(static)\}
\end{array}$$

$(\texttt{instOf} : \texttt{Reference\_type} \times \texttt{Object} \to \texttt{Bool})$

$$\begin{array}{l}
\texttt{instOf}(d, o) = \\
\quad \texttt{case}\ o\ \texttt{of} \\
\qquad \texttt{Class_inst_obj}\,(c_ref, _) \to \texttt{compat}(d, c_ref) \\
\qquad \texttt{Array_obj}\,(n, a, _) \to \texttt{compat}(d, (n, a))
\end{array}$$

We use the function `compat` : `Reference_type` × `Reference_type` → `Bool` which, in turn, uses local functions `compatArray`, and `supers` and `interfaces`. These latter two return the set of superclasses and superinterfaces, respectively, of a class. Since we have not considered interfaces here, we assume for now that `interfaces` returns the empty set.

```
compat(d_t, d_s) =
   case d_s of
```

```
        Class_ref  _ -> d_t in supers(d_s) or d_t in interfaces(d_s)
        Array_type _ -> case d_t of
                          Class_ref  _ -> d_t = javacard.lang.Object
                          Array_type _ -> compatArray(d_t,d_s)

    compatArray(d_t, d_s) =
     let (_, d_1), (_, d_2) = d_t, d_s in
       case d_1 of
        Primitive _ -> d_1 = d_2
          _         -> not primitive(d_2) and compat(d_1, d_2)
```

($\texttt{constant\_pool} : \texttt{Class\_ref} \rightarrow (\texttt{CP\_index} \rightarrow \texttt{CP\_info})$)
$\lambda c : \texttt{Class\_name}.\lambda\langle\_, i\rangle : \texttt{Class\_name} \times \texttt{Index} \,.\, \texttt{let}\ \langle \ldots cp \ldots \rangle = env\_name(pack\_name(c))(c)\ \texttt{in}\ cp(i)$

($\texttt{method\_class} : \texttt{Method\_ref} \rightarrow \texttt{Class\_ref}$)
$\lambda\langle tag, c\_name, sig\rangle : \texttt{Method\_ref} \,.\, c\_name$

($\texttt{super} : \texttt{Class\_ref} \rightarrow \texttt{Class\_ref}$)
$\lambda c : \texttt{Class\_name} \,.\, env\_name(pack\_name(c))\ c \,.\, Super$

($\texttt{super\_invocation} : \texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Bool}$)
$\lambda\langle c, \langle c', \langle m\_name, t\rangle\rangle\rangle : \texttt{Class\_name} \times \texttt{Method\_ref} \,.$
$env\_name(pack\_name(c))(c).Class\_flags(\texttt{super}) \wedge$
$m\_name \neq \texttt{<init>} \wedge$
$\neg env\_name((pack\_name(c'))(c').Methods\_item(\langle m\_name, t\rangle).Method\_flags(\texttt{private}))$

The `super` class flag does not seem to be semantically significant. It exists only for reasons of backward compatibility.

($\texttt{method\_code} : \texttt{Method\_ref} \rightarrow \texttt{Bytecode}$)
$\lambda\langle tag, c\_name, sig\rangle : \texttt{Method\_ref} \,.\, env\_name(pack\_name(c\_name))\ c\_name \,.\, Methods\_item(sig) \,.\, Bytecode$

($\texttt{method\_nargs} : \texttt{Method\_ref} \rightarrow \texttt{Nat}$)
$\lambda\langle c\_name, \langle m\_name, types\rangle\rangle : \texttt{Method\_ref} \,.\, length(types)$

The function *length* returns the length of the list of types.

($\texttt{reference\_type} : \texttt{Type\_code} \times \texttt{Class\_ref} \rightarrow \texttt{Reference\_type}$)
$\lambda\langle i, c\rangle \,.\, f(env(pack\_name(c))(c).Constant\_pool\_item(i))$

We assume a partial coercion function, f, of type `CP_info` $\rightarrow$ `Array_type`, which takes a class name representing an array type, and returns the actual array type.

7 Token Interpretation

In the token format, the data is arranged by packages into CAP files. Each CAP file consists of a number of components.

There are tokens for the various entities — packages, classes, static fields, static methods, instance fields, virtual methods, and instance methods — each with a particular range and scope. Although package tokens should be scoped *within* a particular CAP file, (being indices into the package table which gives an AID), we will assume they are externally visible here.

The *export file* contains a list of tokens for imported entities. We only make use of this in assuming the existence of a token for the class `javacard.lang.Object`.

References to items external to a package are via tokens, which are used to find internal offsets. For example, the class component consists of a list of class information structures, each of which has method tables indexed by tokens that give offsets into the method component, where the method information is found. Internal references use the offsets directly.

7.1 Types

$$[\![\texttt{Package}]\!]_{tok} = \texttt{CAP\_file}$$

$$[\![\texttt{Package\_ref}]\!]_{tok} = \texttt{Package\_tok}$$

It follows that an environment consists of a mapping from package tokens to their corresponding CAP file.

$$env_{tok} : \texttt{Package\_tok} \to \texttt{CAP\_file}$$

A CAP file consists of 11 components, though not all are used for method lookup (or, indeed, the rest of the operational semantics). We just include those components we need.

$$\texttt{CAP\_file} = \texttt{Constant\_pool\_comp} \times \texttt{Class\_comp} \times \texttt{Method\_comp} \times \texttt{Static\_field\_comp} \times \texttt{Descriptor\_comp}$$

The class reference is either an internal offset into the class component of the CAP file of this package, or an external reference composed of a package token and a class token. However, since we need to relate the reference to class names, we will assume that all references come with package information, even though this is superfluous in the case of internal references. Thus we define

$$[\![\texttt{Class\_ref}]\!]_{tok} = \texttt{Package\_tok} \times (\texttt{Class\_tok} + \texttt{Offset})$$

$$[\![\texttt{Field\_ref}]\!]_{tok} = \texttt{Static\_field\_ref} + \texttt{Instance\_field\_ref}$$

$$\texttt{Static\_field\_ref} = [\![\texttt{Class\_ref}]\!]_{\texttt{tok}} \times (\texttt{Static\_field\_tok} + \texttt{Offset})$$

$$\texttt{Instance\_field\_ref} = [\![\texttt{Class\_ref}]\!]_{\texttt{tok}} \times \texttt{Instance\_field\_tok}$$

$$[\![\texttt{Method\_ref}]\!]_{tok} = \texttt{Static\_method\_ref} + \texttt{Virtual\_method\_ref} + \texttt{Super\_method\_ref}$$

$$\texttt{Static\_method\_ref} = [\![\texttt{Class\_ref}]\!]_{\texttt{tok}} \times (\texttt{Static\_method\_tok} + \texttt{Offset})$$

$$\texttt{Virtual\_method\_ref} = [\![\texttt{Class\_ref}]\!]_{\texttt{tok}} \times \texttt{Virtual\_method\_tok}$$

$$\texttt{Super\_method\_ref} = [\![\texttt{Class\_ref}]\!]_{\texttt{tok}} \times \texttt{Virtual\_method\_tok}$$

$$[\![\texttt{Interface\_method\_ref}]\!]_{tok} = [\![\texttt{Class\_ref}]\!]_{\texttt{tok}} \times \texttt{Interface\_method\_tok}$$

$$[\![\texttt{CP\_index}]\!]_{tok} = \texttt{Package\_tok} \times \texttt{Index}$$

Let $\texttt{SMP} = \texttt{SM\_index} \to \texttt{SM\_ref}$ ('supermethod pool').

$$[\![\texttt{Constant\_pool}]\!]_{tok} = [\![\texttt{CP\_index} \to \texttt{CP\_intro}]\!]_{tok} \times \texttt{SMP}$$

$$[\![\texttt{Class}]\!]_{tok} = \texttt{Class\_info}$$

$$\texttt{Class\_comp} = \texttt{Classes} \times \texttt{Interfaces}$$

We ignore the tag and size items.

$$\texttt{Classes} = \texttt{Offset} \to \texttt{Class\_info}$$

$$\begin{aligned}\texttt{Class\_info} \quad = \quad & \texttt{Class\_flags} \times \texttt{Interface\_count} \times \texttt{Super} \times \texttt{Public\_table} \times \texttt{Package\_table} \times \\ & [\![\texttt{Class\_ref}]\!]_{\texttt{tok}}\end{aligned}$$

The class reference is to the class itself.

$$\texttt{Class\_flags} = \texttt{Class\_flag} \to \texttt{Bool}$$

$$\texttt{Class\_flag} = \texttt{Interface} + \texttt{Shareable}$$

Access information is not given by the flags in CAP format. We assume that each `Class_info` structure is labelled with the corresponding class reference. We do not regard `0xFFFF` as an offset.

$$\texttt{Public\_table} = \texttt{Public\_base} \times \texttt{Public\_size} \times (\texttt{Index} \to \texttt{Offset} + \{\texttt{0xFFFF}\})$$

$$\texttt{Package\_table} = \texttt{Package\_base} \times \texttt{Package\_size} \times (\texttt{Index} \to \texttt{Offset})$$

The two method tables each contain a base, size and 'list' of entries. The entries are defined from the *base* to $base + size - 1$ inclusive.

It is convenient to sometimes consider the two method tables together. We write *Class_info . Tables* for the union of the two tables.

We abstract from the details of the method and constant pool components and regard them as mappings from indices to entries. We use `Index` to access elements of lists, and `Offset` to access tables, but since we formalise both as functions, the distinction is not important.

$$\texttt{Constant\_pool\_comp} = [\![\texttt{Constant\_pool}]\!]_{\texttt{tok}}$$

$$[\![\texttt{Pack\_methods}]\!]_{tok} = \texttt{Method\_comp}$$

$$\texttt{Method\_comp} = \texttt{Exception\_handlers} \times \texttt{Methods}$$

$$\texttt{Methods} = (\texttt{Offset} \to \texttt{Method\_info})$$

$$\texttt{Method\_info} = \texttt{Method\_flags} \times \texttt{Maxstack} \times \texttt{Nargs} \times \texttt{Max\_locals} \times \texttt{Bytecode}$$

$$\texttt{Method\_flags} = \texttt{Method\_flag} \to \texttt{Bool}$$

$$\texttt{Method\_flag} = \texttt{Extended} + \texttt{Abstract}$$

$$\texttt{Exception\_handlers} = \texttt{Index} \to \texttt{Exception\_handler\_info}$$

$$[\![\texttt{Pack\_fields}]\!]_{tok} = \texttt{Static\_field\_comp} \times \texttt{Descriptor\_comp}$$

The static field component contains details of the static fields in the classes of a package. The instance fields, on the other hand, only appear in the descriptor component. We assume offsets are into the appropriate list.

$$\texttt{Static\_field\_comp} = \texttt{Array\_init} \times \texttt{Non\_default\_values}$$

$$\texttt{Array\_init} = \texttt{Offset} \to \texttt{Array\_init\_info}$$

$$\texttt{Array\_init\_info} = \texttt{Count} \times (\texttt{Index} \to \texttt{Value})$$

$$\texttt{Non\_default\_values} = \texttt{Offset} \to \texttt{Value}$$

The descriptor component is used for "parsing and verifying" the CAP file. The only use of it in the semantics seems to be to find the types of instance fields for the `new` instruction.

$$\texttt{Descriptor\_comp} = \texttt{Class\_ref} \to \texttt{Class\_descriptor\_info}$$

$$\begin{aligned}\texttt{Class\_descriptor\_info} \quad = \quad & (\texttt{Class\_tok} + \{\texttt{0xFFFF}\}) \times \texttt{Access\_flags} \times \texttt{Offset} \times \\ & \texttt{Interface\_count} \times \texttt{Field\_count} \times \texttt{Method\_count} \times \\ & \texttt{Interfaces} \times \texttt{Field\_descs} \times \texttt{Method\_descs}\end{aligned}$$

Rather than represent arrays as lists, and then define functions to traverse the lists and find an entry with a given index, we choose to represent the arrays directly as functions of the various indices. In the class file, where entities have unique names, the name can be used as an index. In the CAP file, however, it is not so clear what to use as an index since some arrays contain both internal and external items. In the specification [Sun99] the `Fields` item (`Field_descs` here) is a list of field descriptors, of the form

$$(\texttt{Token} + \{\texttt{0xFFFF}\}) \times \texttt{Access\_flags} \times [\![\texttt{Field\_ref}]\!]_{\texttt{tok}} \times \texttt{Type\_desc}$$

The field descriptor items contain a token (or `0xFFFF`) and a field reference. Presumably, the field reference gives the offset for static field references, and repeats the token for instance field references. We reformulate this in a triple $\texttt{Field\_descs} = \texttt{Instance\_fields} \times \texttt{Pub\_static\_fields} \times \texttt{Pack\_static\_fields}$ where

$$\texttt{Instance\_fields} = \texttt{Token} \to \texttt{Access\_flags} \times [\![\texttt{Field\_ref}]\!]_{\texttt{tok}} \times \texttt{Type\_desc}$$

$$\texttt{Pub\_static\_fields} = \texttt{Token} \to \texttt{Access\_flags} \times [\![\texttt{Field\_ref}]\!]_{\texttt{tok}} \times \texttt{Type\_desc}$$

$$\texttt{Pack\_static\_fields} = [\![\texttt{Field\_ref}]\!]_{\texttt{tok}} \to \texttt{Access\_flags} \times \texttt{Type\_desc}$$

Thus, the transformation should really have another pass, in which the functions would be flattened into arrays, and the true offsets calculated using the size of the various entries. We ignore this here.

We do not give the details for the `Method_descs` item since it is not used in the semantics.

Finally,

$$[\![\texttt{Type\_code}]\!]_{tok} = \texttt{Nat} \times ([\![\texttt{CP\_index}]\!]_{tok} + \{0\})$$

7.2 Auxiliary Functions

The lookup function takes a class reference (the declared class), a method reference (in the actual class), and returns the reference to the class where the code is defined, together with the bytecode itself.

We use several local auxiliary functions:

$$\texttt{class\_info} : \texttt{Class\_ref} \to \texttt{Class\_info}$$

$$\texttt{class\_info}' : \texttt{Package} \times \texttt{Class\_ref} \to \texttt{Class\_info}$$

$$\texttt{methods\_array} : \texttt{Class\_ref} \to (\texttt{Offset} \to \texttt{Method\_info})$$

$$\texttt{same\_package} : \texttt{Package\_ref} \times \texttt{Package\_ref} \to \texttt{bool}$$

$\texttt{same\_package}(c\_ref, \texttt{c}'\texttt{\_ref}) =$
$\quad \texttt{let}(\texttt{p\_tok}, \_), (\texttt{p}'\texttt{\_tok}) = c\_ref, c\_ref' \texttt{ in}$
$\quad\quad \texttt{p\_tok} = \texttt{p}'\texttt{\_tok}$

For a given class reference, the function `class_info` finds the corresponding class information structure in the global environment.

```
class_info (c_ref) =
let (..., cp_comp, class_comp, method_comp, ...) : CAP_file =

case c_ref of
   Int (int_pack_tok, _) -> env_tok(int_pack_tok)
   Ext (ext_pack_tok, _) -> env_tok(ext_pack_tok)
in

case c_ref of
   Int (_, offset) -> class_comp.Classes (offset)
   Ext (_, class_tok)    ->
      let class_offset = class_offset(class_tok)
         in class_comp.Classes (offset)
```

We define a variant, $\texttt{class\_info}'$, which returns the class information structure in a particular CAP file.

```
methods_array (class_ref) =
let (..., method_comp, ...) : CAP_file =
 case class_ref of
    Int (int_pack_tok, _) -> env_tok(int_pack_tok)
    Ext (ext_pack_tok, _) -> env_tok(ext_pack_tok)
 in
 method_comp.Methods
```

We assume the existence of several functions for resolving external tokens to internal offsets.[7]

$$\texttt{class\_offset} : \texttt{Package\_tok} \times \texttt{Class\_tok} \to \texttt{Offset}$$

$$\texttt{static\_field\_offset} : \texttt{Package\_tok} \times \texttt{Class\_tok} \times \texttt{Field\_tok} \to \texttt{Offset}$$

$$\texttt{method\_offset} : \texttt{Package\_tok} \times \texttt{Class\_tok} \times (\texttt{Virtual\_method\_tok} + \texttt{Static\_method\_tok}) \to \texttt{Offset}$$

We extend these definitions in the obvious way to take arbitrary references.

The main steps of `lookup_tok` for virtual methods are:

1. Get method array for the package of the actual class.
2. Get class information for the actual class.

[7] It is not clear how these should be implemented. One possibility is through the descriptor component; another is to use the export component.

```
lookup_tok (act_class_ref, (tag, dec_class_ref, method_tok)) =

case tag of

CONSTANT_virtual_method_ref, CONSTANT_super_method_ref ->


let methods = method_array (act_class_ref)
in
let (..., super, public_base, public_table, package_base, package_table,...)
                                                     : Class_info =
    class_info(act_class_ref)
in
if method_tok div 128 = 0 then /* public */
   if method_tok >= public_base then
      let method_offset = public_table[method_tok - public_base]
        in
        if method_offset <>  0xFFFF
        then (act_class_ref, methods[method_offset].Bytecode)
        else /* look in superclass */
           lookup_tok(super, (tag, dec_class_ref, method_tok))
   else /* look in superclass */
      lookup_tok(super, (tag, dec_class_ref, method_tok))
else /* package */
   if method_tok >= package_base /\
      (same_package(dec_class_ref, act_class_ref)
          \/ tag = CONSTANT_super_method_ref)
   then
      let method_offset = package_table[method_tok mod 128 - package_base]
        in (act_class_ref, methods[method_offset].Bytecode)
   else /* look in superclass */
      lookup_tok(super, (tag, dec_class_ref, method_tok))
```

3. If public:
   if defined then *get info* else *lookup super*
   If package:
   if defined $\wedge$ visible then *get info* else *lookup super*

$(\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \to \texttt{Class\_ref} \times \texttt{Bytecode})$
$\quad [\![\texttt{lookup}]\!]_{tok} = \texttt{lookup\_tok}$

$(\texttt{lookup\_int} : \texttt{Class\_ref} \times \texttt{Interface\_method\_ref} \to \texttt{Class\_ref} \times \texttt{Bytecode})$
$\quad [\![\texttt{lookup\_int}]\!]_{tok} = \texttt{lookup\_int\_tok}$ — *not defined here*

$(\texttt{static\_val} : \texttt{Field\_ref} \to \texttt{Value})$
$\quad \lambda\langle\langle p, c\rangle, f\rangle : \texttt{Field\_ref}\,.$
$\quad\quad \texttt{let}\ \mathit{offset} = \texttt{case}\ f\ \texttt{of}$
$\quad\quad\quad \texttt{Int}\ i \to i$
$\quad\quad\quad \texttt{Ext}\ tok \to \texttt{static\_field\_offset}(p, c, tok)$
$\quad\quad \texttt{in}$
$\quad\quad\quad env\_tok(p).Static\_field\_comp.Non\_default\_values(\mathit{offset})$

$(\texttt{update} : \texttt{Field\_ref} \times \texttt{Value} \to \texttt{Statechange})$

$$
\begin{array}{l}
\texttt{update}(\langle\langle p,c\rangle, f\rangle, v) = \\
\quad \lambda\langle e,h\rangle. \\
\quad\quad \langle\lambda p'. \\
\quad\quad\quad \texttt{if } p' = p \texttt{ then} \\
\quad\quad\quad\quad \texttt{let } \langle\_,\_,\_,\langle Array\_init, Ndv\rangle,\_\rangle = e(p) \texttt{ in} \\
\quad\quad\quad\quad \texttt{let } offset = \texttt{case } f \texttt{ of} \\
\quad\quad\quad\quad\quad \texttt{Int } i \to i \\
\quad\quad\quad\quad\quad \texttt{Ext } tok \to static\_field\_offset(p,c,tok) \\
\quad\quad\quad\quad \texttt{in} \\
\quad\quad\quad\quad\quad \langle\_,\_,\_,\langle Array\_init, Ndv + \{offset \mapsto v\}\rangle,\_\rangle \\
\quad\quad\quad \texttt{else } e(p') \\
\quad\quad\quad\quad\quad\quad\quad , h\rangle
\end{array}
$$

$(\texttt{constant\_pool} : \texttt{Class\_ref} \to (\texttt{CP\_index} \to \texttt{CP\_info}))$

$$
\begin{array}{l}
[\![\texttt{constant\_pool}]\!]_{tok} = \\
\quad \lambda\langle p\_tok, \_\rangle.\lambda x.\texttt{let } \langle cp, smp\rangle = env\_tok(p\_tok).Constant\_pool\_comp \texttt{ in} \\
\quad\quad \texttt{case } x \texttt{ of} \\
\quad\quad\quad \texttt{Index}\ \langle\_,i\rangle \to cp(i) \\
\quad\quad\quad \texttt{SM}\ \langle\_,i\rangle \to smp(i)
\end{array}
$$

$(\texttt{instance\_fields} : \texttt{Class\_ref} \to (\texttt{Field\_ref} \to \texttt{Value}))$

$$
\begin{array}{l}
\texttt{instance\_fields}(c\_ref) = \\
\quad \texttt{let } (p\_tok, c\_tok) = c\_ref \texttt{ in} \\
\quad\quad\quad fields = env\_tok(p\_tok).Descriptor\_comp.Fields.Instance\_fields \texttt{ in} \\
\quad\quad\quad super = \texttt{class\_info}(c\_ref).Super \texttt{ in} \\
\quad \texttt{if } c\_tok = \texttt{javacard.lang.Object\_token}) \texttt{ then} \\
\quad\quad\quad \texttt{instance\_fields1}(c\_ref, fields) \\
\quad \texttt{else instance\_fields}(super) \cup \texttt{instance\_fields1}(c\_ref, fields)
\end{array}
$$

We assume the existence of a token, `javacard.lang.Object_token`. The function *default* computes the default values for each type.

$$
\begin{array}{l}
\texttt{instance\_fields1}(c\_ref, fields) = \\
\quad\quad\quad \{\ (\ \langle c\_ref, f\_tok\rangle \mapsto default(d)\ )\ \mid\ \exists f\_tok\,.\,fields(f\_tok) = \langle flags, f\_ref, d\rangle\}
\end{array}
$$

$(\texttt{instOf} : \texttt{Reference\_type} \times \texttt{Object} \to \texttt{Bool})$

The same as in the name interpretation, except for the definition of `supers` and `interfaces`.

$(\texttt{super} : \texttt{Class\_ref} \to \texttt{Class\_ref})$

$\lambda c\_ref : \texttt{Class\_ref}\ .\ \texttt{class\_info}(c\_ref)\ .\ Super$

$(\texttt{super\_invocation} : \texttt{Class\_ref} \times \texttt{Method\_ref} \to \texttt{Bool})$

$\lambda\langle\_,\langle tag,\_,\_\rangle\rangle : \texttt{Class\_ref} \times \texttt{Method\_Ref}\ .\ tag = \texttt{CONSTANT\_super\_method\_ref}$

$(\texttt{method\_class} : \texttt{Method\_ref} \to \texttt{Class\_ref})$

$\lambda\langle c\_ref, \_\rangle : \texttt{Method\_ref}\ .\ c\_ref$

$(\mathtt{method\_code} : \mathtt{Method\_ref} \to \mathtt{Bytecode})$
$\quad \mathtt{method\_code}(m\_ref : \mathtt{Method\_ref}) =$
$\quad\quad \mathtt{case}\ m\_ref\ \mathtt{of}$
$\quad\quad\quad \mathtt{CONSTANT\_virtual\_method\_ref}(c\_ref, m\_tok) \to$
$\quad\quad\quad\quad \mathtt{let}\ \langle \_, \_, \_, \_, \_, public\_table, package\_table, \_\rangle = class\_info(cref)\ \mathtt{in}$
$\quad\quad\quad\quad \mathtt{let}\ \langle \_, \_, method\_comp\rangle = env\_tok(pack\_tok)\ \mathtt{in}$
$\quad\quad\quad\quad \mathtt{let}\ offset = tables[m\_tok]$
$\quad\quad\quad\quad \mathtt{in}\ (method\_comp.Methods[offset]).Bytecode$
$\quad\quad\quad \mathtt{CONSTANT\_static\_method\_ref}(p\_tok, c, loc) \to$
$\quad\quad\quad\quad \mathtt{let}\ offset =$
$\quad\quad\quad\quad\quad \mathtt{case}\ loc\ \mathtt{of}$
$\quad\quad\quad\quad\quad\quad m\_tok \to method\_offset(p\_tok, c, m\_tok)$
$\quad\quad\quad\quad\quad\quad offset' \to offset'$
$\quad\quad\quad\quad\quad \mathtt{in}\ (method\_comp.Methods[offset]).Bytecode$

$(\mathtt{method\_nargs} : \mathtt{Method\_ref} \to \mathtt{Nat})$
$\quad \lambda m\_ref : \mathtt{Method\_ref}\ .\ \mathtt{method\_info}(m\_ref)\ .\ Nargs$

We use a local function `method_info` to return the `Method_info` corresponding directly to a method reference.

$(\mathtt{reference\_type} : \mathtt{Type\_code} \times \mathtt{Class\_ref} \to \mathtt{Reference\_type})$

$$
\lambda\langle n, i, \langle p, c\rangle\rangle\ .\ \mathtt{case}\ n\ \mathtt{of}
\begin{array}{rcl}
0 & \to & f(env\_tok(p)\ .\ Constant\_pool\_comp(i)) \\
10 & \to & \mathtt{Array\ Boolean} \\
11 & \to & \mathtt{Array\ Byte} \\
12 & \to & \mathtt{Array\ Short} \\
13 & \to & \mathtt{Array\ Int} \\
14 & \to & \mathtt{Array}\ f(env\_tok(p)\ .\ Constant\_pool\_comp(i))
\end{array}
$$

The function $f : \mathtt{CP\_info} \to \mathtt{Reference\_type}$ is used to convert the type:

$\lambda x : \mathtt{CP\_info}\ .\ \mathtt{case}\ x : \mathtt{CP\_info}\ \ \mathtt{of}$
$\quad\quad \mathtt{Class\_ref}\ c \to (\mathtt{Class\_ref}\ c\ :\ \mathtt{Reference\_type})$

8 Formalisation of Equivalence

We now formalise the equivalence between the class and CAP formats as a family of relations, $\{R_\theta : [\![\theta]\!]_{name} \leftrightarrow [\![\theta]\!]_{tok}\}_{\theta \in Abstract\_type}$ indexed by abstract type, θ. In fact, it is convenient to cheat a little by defining relations for certain types that do not correspond to any abstract types. The idea is that there is a fixed family of relations such that $x\ R_\theta\ y$ when y is a *possible* transformation of x. The relations are not necessarily total, *i.e.* for some $x : [\![\theta]\!]_{name}$, there may not be a y such that $x\ R_\theta\ y$. We make no restrictions on the relation domains.[8]

We make various suppositions of the well-formedness of the input. For example, we assume that the class hierarchy is well-founded and that `javacard.lang.Object` is the top. We make no assumptions of well-formedness for CAP files, however. The only notion of well-formedness for a CAP file is that it is the result of transforming a collection of well-formed class files. Formally, the relations are defined as a mutually inductive collection of constraints, R_θ, for each type θ, where the types, θ, are given by the grammar:

[8] For example, arrays are not in the domain of $R_{\mathtt{Class\_ref}}$. This is not important for the proof of correctness. However, this would have to be taken into account for the development of an algorithm.

$$\begin{array}{lcl}
\gamma & ::= & \texttt{Bool} \mid \texttt{Nat} \mid \texttt{Object_ref} \mid \texttt{Boolean} \mid \texttt{Byte} \mid \texttt{Short} \mid \texttt{Value} \mid \texttt{Word} \\
\iota & ::= & \texttt{Package_ref} \mid \texttt{Class_ref} \mid \texttt{Static_field_ref} \mid \texttt{Instance_field_ref} \mid \texttt{Static_method_ref} \mid \\
 & & \texttt{Interface_method_ref} \mid \texttt{Virtual_method_ref} \mid \texttt{SM_ref} \\
\kappa & ::= & \texttt{CP_index} \mid \texttt{SM_index} \mid \texttt{CP_info} \mid \texttt{Method_info} \mid \texttt{Type_code} \mid \texttt{Package} \mid \texttt{Class} \mid \\
 & & \texttt{Constant_pool} \mid \texttt{Pack_methods} \mid \texttt{Pack_fields} \\
\theta & ::= & \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \to \theta'
\end{array}$$

There are two sources of underspecification here. On the one hand, the relations really can be non-functional. On the other, there is a choice for what some of the relations are. For example, $R_{\texttt{Class\_ref}}$ is *some* bijection satisfying certain constraints. The relations between the 'large' structures, however, are completely defined in terms of those between smaller ones.

We first give the standard definitions of logical relations for the type constructors used here. These are used throughout the definition of the transformation. Then there are two parts to the transformation itself: the tokenisation, defined as the relations R_ι, and the 'componentisation', defined as the R_κ.

8.1 Logical Relations

There are standard definitions of $R_{\theta\times\theta'}$, $R_{\theta\to\theta'}$ and $R_{\theta+\theta'}$ in terms of R_θ and $R_{\theta'}$. In addition, for each basic type γ, we have $R_\gamma = id_\gamma$.

$$a\ R_\gamma\ a' \iff a = a'$$

$$f\ R_{\theta\to\theta'}\ f' \iff \forall a\ R_\theta\ a'\ .\ fa\ R_{\theta'}\ f'a'$$

In general, functions are partial. Thus if fa is defined and $a\ R_\theta\ a'$, then $f'a'$ must be defined.

$$\langle a, b\rangle\ R_{\theta\times\theta'}\ \langle a', b'\rangle \iff a\ R_\theta\ a' \wedge b\ R_\theta\ b'$$

$$a\ R_{\theta+\theta'}\ a' \iff \begin{array}{l}(\exists b, b'\ .\ a = \texttt{Theta}\ b\ \wedge\ a' = \texttt{Theta}\ b'\ \wedge\ b\ R_\theta\ b') \vee \\ (\exists c, c'\ .\ a = \texttt{Theta}'\ c\ \wedge\ a' = \texttt{Theta}'\ c'\ \wedge\ c\ R_{\theta'}\ c')\end{array}$$

Moreover, there is an obvious definition for lists:

$$[\,]\ R_{\theta^*}\ [\,]$$

$$a :: as\ R_{\theta^*}\ a' :: as' \iff a\ R_\theta\ a' \wedge as\ R_{\theta^*}\ as'$$

Strictly speaking, because the types are mutually recursive, we should define the relations recursively, but we will gloss over this point. As an example of a derived relation, it follows that $R_{\texttt{Heap}}$ is defined as:

$$heap_{name}\ R_{\texttt{Heap}}\ heap_{tok} \iff \forall r : \texttt{Object\_ref}\ .\ heap_{name}(r)\ R_{\texttt{Object}}\ heap_{tok}(r)$$

where $R_{\texttt{Object}}$ is defined in terms of $R_{\texttt{Class\_ref}}$.

8.2 Tokenisation

The relations R_ι represent the tokenisation of items. The general idea is to set up relations between the names and tokens assigned to the various entities, subject to certain constraints described in the specification.

In order to account for token scope, we relate names to tokens paired with the appropriate context information. For example, class tokens are scoped within a package, so the relation $R_{\texttt{Ext\_class\_ref}}$ is between individual class names, and pairs of package and class tokens. We must add a condition, therefore, to ensure that the package token corresponds to the package name of this class name.

We assume that each of these relations is a bijection (with one exception to account for the copying of virtual methods in the token format). Formally, a relation, R, is bijective when it is functional in both directions.

$$a\,R\,b\ \wedge\ a'\,R\,b\ \Rightarrow\ a = a'$$

and vice versa,

$$a\,R\,b\ \wedge\ a\,R\,b'\ \Rightarrow\ b = b'$$

However, relations are only bijective modulo the equivalence between internal and external references so we modify this to:

$$a\,R\,b \;\wedge\; a'\,R\,b \;\Rightarrow\; a = a'$$

$$a\,R\,b \;\Rightarrow\; (a\,R\,b' \iff \texttt{Equiv}(b, b'))$$

where equivalence, `Equiv`, of class references is defined as the reflexive symmetric closure of:

$$\texttt{Equiv}(\langle p\_tok, \mathit{offset}\rangle, \langle p\_tok, c\_tok\rangle) \iff \texttt{class\_offset}(p\_tok, c\_tok) = \mathit{offset}$$

We will say that the relation is 'externally bijective'. The second condition contains two parts: that the relation is injective modulo `Equiv`, and that it is closed under `Equiv`. We say that R is an *external bijection* when these conditions hold. We extend the definition of `Equiv` and external bijection to the other references. For example,

$$\texttt{Equiv}(\langle c\_ref, f\_tok\rangle, \langle c'\_ref, f'\_tok\rangle) \iff f\_tok = f'\_tok \wedge \texttt{Equiv}(c\_ref, c'\_ref)$$

We cannot override static methods, but can override interface and virtual methods. The only bearing this has on the relations is in certain constraints for the tokenisation of overridden virtual methods.

The tokenisation process assigns tokens to *external* class references, static fields and static methods, and to all instance fields, virtual methods and interface methods.

These relations are defined with respect to the environment (in name format). We use a number of abbreviations for extracting information from the environment. We write $c < c'$ for the subclass relation (*i.e.* the transitive closure of the direct subclass relation) and $\leq$ for its reflexive closure. In the token interpretation this is modulo `Equiv`. We write $m\_tok \in c\_ref$ when a method with token m_tok is declared in the class with reference c_ref. That is, $m\_tok \in dom(Tables)$ of $class\_info(c\_ref)$ (defined on p. 20), and $pack\_name(c)$ for the package name of the class named c.

We define functions for accessing flags:

$$\texttt{Class\_flag}(c\_name, flag) = env\_name(c\_name)\,.\,Class\_flags(flag)$$

$$\texttt{Field\_flag}(c\_name, f\_name, flag) = env\_name(c\_name)\,.\,Fields(f\_name)\,.\,Field\_flags(flag)$$

The tokenisation uses the notion of *external visibility*.

$$\texttt{Externally\_visible}(c\_name) = \texttt{Class\_flag}(c\_name, Public)$$

$$\texttt{Externally\_visible}(c\_name, f\_name) = \texttt{Class\_flag}(c\_name, Public) \wedge \texttt{Field\_flag}(c\_name, f\_name, Public)$$

We will also write $public(sig)$ and $package(sig)$.

The relations for `Instance_field_ref`, `Virtual_method_ref` and `Interface_method_ref` use `Class_ref`, defined in the next section.

(`Package_ref`) As mentioned above, we take package tokens to be externally visible. The relation $R_{\texttt{Package\_ref}}$ is simply defined as any bijection between package names and tokens.

(`Ext_class_ref`) A bijection such that: $c\_name\ R_{\texttt{Ext\_class\_ref}}\ (p\_tok, c\_tok) \Rightarrow \texttt{Externally\_visible}(c\_name) \wedge pack\_name(c\_name)\ R_{\texttt{Package\_ref}}\ p\_tok$

(`Instance_field_ref`) Package tokens must be higher than public tokens. Note that, in contrast to virtual method tokens, public and package tokens are drawn from the *same* namespace, and so this condition does not follow automatically.

$\langle c\_name, sig\rangle\ R_{\texttt{Instance\_field\_ref}}\ \langle c\_ref, f\_tok\rangle \wedge$
$\langle c'\_name, sig'\rangle\ R_{\texttt{Instance\_field\_ref}}\ \langle c'\_ref, f'\_tok\rangle \wedge public(sig) \wedge package(sig') \Rightarrow f\_tok < f'\_tok$

The relation respects $R_{\texttt{Class\_ref}}$:
$\langle c\_name, sig\rangle\ R_{\texttt{Instance\_field\_ref}}\ \langle c\_ref, f\_tok\rangle \Rightarrow c\_name\ R_{\texttt{Class\_ref}}\ c\_ref$

Since the relation uses $R_{\texttt{Class\_ref}}$ it is not bijective. However, it is functional from names to tokens, whereas in the other direction we have:

$$\langle c\_name, sig\rangle\ R_{\texttt{Instance\_field\_ref}}\ \langle c\_ref, f\_tok\rangle \;\wedge\; \langle c\_name, sig\rangle\ R_{\texttt{Instance\_field\_ref}}\ \langle c'\_ref, f'\_tok\rangle \Rightarrow \texttt{Equiv}(\langle c\_ref, f\_tok\rangle, \langle c'\_ref, f'\_tok\rangle)$$

(`Ext_static_field_ref`) An external bijection such that: $\langle c\_name, f\_name\rangle\ R_{\texttt{Ext\_static\_field\_ref}}\ \langle c\_ref, f\_tok\rangle \Rightarrow$ $\texttt{Externally\_visible}(c\_name, f\_name) \wedge \texttt{Field\_flag}(c\_name, f\_name, Static) \wedge c\_name\ R_{\texttt{Ext\_class\_ref}}\ c\_ref$

(`Ext_static_method_ref`) An external bijection such that: $\langle c\_name, sig\rangle\ R_{\texttt{Ext\_static\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \Rightarrow$ $\texttt{Externally\_visible}(c\_name, sig) \wedge \texttt{Method\_flag}(c\_name, sig, Static) \wedge c\_name\ R_{\texttt{Ext\_class\_ref}}\ c\_ref$

(`Virtual_method_ref`) This is not a bijection because of the possibility of copying. Although 'from names to tokens' we do have:

$$\langle c\_name, sig\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \langle c'\_name, sig'\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \Rightarrow c\_name = c'\_name \wedge sig = sig'$$

for a converse we have:

$$\langle c\_name, sig\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \langle c\_name, sig\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle c'\_ref, m'\_tok\rangle \Rightarrow (c\_ref \leq c'\_ref \vee c'\_ref \leq c\_ref) \wedge m\_tok = m'\_tok$$

The first condition says that if a method overrides a method implemented in a superclass, then it must take the same token. Restrictions on the language means that overriding cannot change the method modifier from public to package or vice versa.
$\langle c\_name, sig\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge$
$\langle c'\_name, sig\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle c'\_ref, m'\_tok\rangle \wedge$
$(c'\_name < c\_name \wedge (package(sig) \Rightarrow same\_package(c\_name, c'\_name))) \Rightarrow m\_tok = m'\_tok$
The second condition says that the (public) tokens for introduced methods must have higher token numbers that those in the superclass. We assume a predicate, *new_method*, which holds of a method signature and class name when the method is defined in the class, but not in any superclass.

$$public(sig) \wedge new\_method(sig, c\_name) \wedge (c\_name, sig)\ R_{\texttt{Virtual\_method\_ref}}\ (c\_ref, m\_tok) \Rightarrow \forall m'\_tok \in super(c\_ref).m\_tok > m'\_tok$$

Package tokens for introduced methods are similarly numbered, if the superclass is in the same package, but from 0 otherwise.

$$package(sig) \wedge new\_method(sig, c\_name) \wedge (c\_name, sig)\ R_{\texttt{Virtual\_method\_ref}}\ (c\_ref, m\_tok) \wedge same\_package(c\_name, super(c\_name)) \Rightarrow \forall m'\_tok \in super(c\_ref).m\_tok > m'\_tok$$

The third condition says that public tokens are in the range 0 to 127, and package tokens in the range 128 to 255.

$\langle c\_name, sig\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \Rightarrow (public(sig) \Rightarrow 0 \leq m\_tok \leq 127) \wedge (package(sig) \Rightarrow 128 \leq m\_tok \leq 255)$

The specification [Sun99] also says that tokens must be contiguous but we will not enforce this.

(`Interface_method_ref`) $\langle c\_name, sig\rangle\ R_{\texttt{Interface\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \Rightarrow c\_name\ R_{\texttt{Class\_ref}}\ c\_ref$

8.3 Componentisation

The relations in the previous section formalise the correspondence between named and tokenised entities. When creating the CAP file components, all the entities are converted, including the package visible ones. Thus at this point we define $R_{\texttt{Class\_ref}}$, $R_{\texttt{Static\_field\_ref}}$ and $R_{\texttt{Static\_method\_ref}}$, as relations between named items and either external tokens or internal references, subject to coherence constraints.

We must ensure that if a name corresponds to both an external token and to an internal offset, then the token and the offset correspond to the same entity. There are two ways we could ensure this. One possibility is, for example, to use the function $\texttt{class\_info} : \texttt{Class\_ref} \rightarrow \texttt{Class\_info}$ with the constraint:

$$c\_name\ R_{\texttt{Class\_ref}}\ \langle p\_tok, c\_tok\rangle \wedge c\_name\ R_{\texttt{Class\_ref}}\ \langle p\_tok, offset\rangle \Rightarrow class\_info(\langle p\_tok, c\_tok\rangle) = class\_info(\langle p\_tok, offset\rangle)$$

The other possibility is to use the offset function $\texttt{class\_offset} : \texttt{Package\_tok} \times \texttt{Class\_tok} \rightarrow \texttt{Offset}$ which returns the internal offset corresponding to an external token, and then *define* $R_{\texttt{Class\_ref}}$ from this and $R_{\texttt{Ext\_class\_ref}}$, and this is the solution we choose here. Clearly, therefore, $R_{\texttt{Class\_ref}}$ is not a bijection.

(`Class_ref`) We define $R_{\texttt{Class\_ref}}$ as an external bijection which respects $R_{\texttt{Ext\_class\_ref}}$, that is, such that

$$c\_name \; R_{\texttt{Class\_ref}} \; (p\_tok, c\_tok) \iff c\_name \; R_{\texttt{Ext\_class\_ref}} \; (p\_tok, c\_tok)$$

(`Static_field_ref`)

$\langle c\_name, f\_name \rangle \; R_{\texttt{Static\_field\_ref}} \; \langle p\_tok, c\_tok, f\_tok \rangle \iff$
$\quad \langle c\_name, f\_name \rangle \; R_{\texttt{Ext\_static\_field\_ref}} \; \langle p\_tok, c\_tok, f\_tok \rangle$

and

$\langle c\_name, f\_name \rangle \; R_{\texttt{Static\_field\_ref}} \; \langle p\_tok, c\_tok, \mathit{offset} \rangle \iff$
$\exists f\_tok . \mathit{offset} = \texttt{static\_field\_offset}(p\_tok, c\_tok, f\_tok) \wedge$
$\quad \langle c\_name, f\_name \rangle \; R_{\texttt{Static\_field\_ref}} \; \langle p\_tok, c\_tok, f\_tok \rangle$

(`Static_method_ref`)

The relation $R_{\texttt{Static\_method\_ref}}$ is defined similarly, using the function `method_offset`.

$\langle c\_name, m\_name \rangle \; R_{\texttt{Static\_method\_ref}} \; \langle p\_tok, c\_tok, m\_tok \rangle \iff$
$\quad \langle c\_name, m\_name \rangle \; R_{\texttt{Ext\_static\_method\_ref}} \; \langle p\_tok, c\_tok, m\_tok \rangle$

and

$\langle c\_name, m\_name \rangle \; R_{\texttt{Static\_method\_ref}} \; \langle p\_tok, c\_tok, \mathit{offset} \rangle \iff$
$\exists m\_tok . \mathit{offset} = \texttt{method\_offset}(p\_tok, c\_tok, m\_tok) \wedge$
$\quad \langle c\_name, m\_name \rangle \; R_{\texttt{Static\_method\_ref}} \; \langle p\_tok, c\_tok, m\_tok \rangle$

The three 'big' components are the constant pool, method, and class components. We mainly limit our definition of equivalence to these, though also consider the static field and descriptor components.

(`CP_index`) Define $[\![\texttt{CP\_index}]\!]_{tok} = \texttt{Package\_tok} \times \texttt{Index}$.

A bijection such that $\langle c\_name, i \rangle \; R_{\texttt{CP\_index}} \; \langle p\_tok, i' \rangle \Rightarrow pack\_name(c\_name) \; R_{\texttt{Package\_ref}} \; p\_tok$

(`CP_info`) We have defined $\texttt{CP\_info} = \texttt{Class\_ref} + \texttt{Method\_ref} + \texttt{Interface\_method\_ref} + \texttt{Field\_ref}$. We must define how the specific field and method references in the CAP file correspond to those in the class file.

(`Field_ref`) $f\_ref \; R_{\texttt{Field\_ref}} \; \langle tag, f'\_ref \rangle \iff$
$\texttt{Field\_flag}(f\_ref, Static) \wedge tag = CONSTANT\_StaticFieldRef \wedge f\_ref \; R_{\texttt{Static\_field\_ref}} \; f'\_ref$
$\vee$
$\neg\texttt{Field\_flag}(f\_ref, Static) \wedge tag = CONSTANT\_InstanceFieldRef \wedge f\_ref \; R_{\texttt{Instance\_field\_ref}} \; f'\_ref$

(`Method_ref`) A method reference in the class file can become either a static, super or virtual method reference. Super method references also use virtual tokens.

We use predicates, `static` and `super`, to determine whether a method reference in the class file can correspond to a static or super method invocation. If `static` holds, then the method must be a static method reference. However, the `super` predicate just indicates the possibility that a method could be a super reference. If it is called from `invokespecial`, this will be the case, but it will be a virtual method reference when called from `invokevirtual`. Let us write cp for $[\![\texttt{constant\_pool}]\!]_{tok}(c\_ref)$, where c_ref is the relevant class reference.

$\texttt{super}(m\_ref) \iff \exists i \; . \; \texttt{invokespecial} \; i \in code \wedge cp(i) = m\_ref$
The code can appear in any package.

$m\_ref \; R_{\texttt{Method\_ref}} \; \langle tag, m'\_ref \rangle \iff$
$\texttt{static}(m\_ref) \wedge tag = CONSTANT\_StaticMethodRef \wedge m\_ref \; R_{\texttt{Static\_method\_ref}} \; m'\_ref$
$\vee$
$\neg\texttt{static}(m\_ref) \wedge \texttt{super}(m\_ref) \wedge$
$tag = CONSTANT\_SuperMethodRef \wedge m\_ref \; R_{\texttt{Virtual\_method\_ref}} \; m'\_ref$
$\vee$
$tag = CONSTANT\_VirtualMethodRef \wedge m\_ref \; R_{\texttt{Virtual\_method\_ref}} \; m'\_ref$

(SM_index)

A bijection from the CP_index such that the entry is potentially a supermethod, that is, the $i : \texttt{CP\_index}$ such that $\texttt{super}(cp(i))$.

(SM_ref)

Relates virtual method references to the corresponding supermethod references.

Since $\texttt{SMP} = \texttt{SM\_index} \to \texttt{SM\_ref}$ ('supermethod pool'), the definition of $R_{\texttt{SMP}}$ follows. Bytecode is defined using invokevirtual SMP_index.

(Method_info)

We only treat certain parts of the method information here:

$$\begin{aligned}
&\langle flags, sig, _, _, maxstack, maxlocals, code, _\rangle \ R_{\texttt{Method_info}} \ \langle flags', maxstack', nargs', maxlocals', code'\rangle \\
&\iff \\
&\quad flags \ R_{\texttt{Method_flags}} \ flags' \ \wedge \\
&\quad maxstack = maxstack' \ \wedge \\
&\quad size(sig) = nargs' \ \wedge \\
&\quad maxlocals = maxlocals' \ \wedge \\
&\quad code \ R_{\texttt{Bytecode}} \ code'
\end{aligned}$$

Flags are used less in the CAP format than in class files. Instead, access information is implicit in the use of tokens. For class flags, we simply have

$$\texttt{Interface} \ R_{\texttt{Class\_flag}} \ \texttt{Interface}$$

Although, there is also a Shareable flag in the CAP format, we assume that no constraints are placed on this by a class file.

(Type_code)

Bytecode verification ensures that the constant pool entry must be a reference type.

$$\begin{aligned}
&i \ R_{\texttt{Type_code}} \ \langle n, i'\rangle \iff \\
&\qquad \texttt{let} \ \langle c, _\rangle = i \ \texttt{in} \\
&\qquad \texttt{let} \ cp_entry = \texttt{constant_pool}(c)(i) \ \texttt{in} \\
&\qquad \exists rt : \texttt{Reference_type}. \neg \texttt{Array}(rt) \ . \\
&\qquad\quad \begin{array}{lllllll}
(n = 0 & \wedge & i \ R_{\texttt{CP_index}} \ i' & \wedge & cp_entry = rt) & \vee \\
(n = 10 & \wedge & i' = 0 & \wedge & cp_entry = \texttt{Array Boolean}) & \vee \\
(n = 11 & \wedge & i' = 0 & \wedge & cp_entry = \texttt{Array Byte}) & \vee \\
(n = 12 & \wedge & i' = 0 & \wedge & cp_entry = \texttt{Array Short}) & \vee \\
(n = 13 & \wedge & i' = 0 & \wedge & cp_entry = \texttt{Array Int}) & \vee \\
(n = 14 & \wedge & i \ R_{\texttt{CP_index}} \ i' & \wedge & cp_entry = \texttt{Array rt})
\end{array}
\end{aligned}$$

We relate individual classes, but methods and constant pools are grouped together. Thus the name interpretation is all the information in one package and so, for example, $[\![\texttt{Pack\_methods}]\!]_{name} : \texttt{Class\_name} \to \texttt{Methods\_item}$ is the 'set' of method data for all classes.

We use the auxiliary function, $\texttt{method\_offset} : \texttt{Package\_tok} \times \texttt{Class\_tok} \times \texttt{Method\_tok} \to \texttt{Offset}$. This is because the method information is spread between the two components in the token format. The relation $R_{\texttt{Class}}$ ensures that a named method corresponds to a particular offset, and $R_{\texttt{Pack\_methods}}$ ensures that the entry at this offset is related by $R_{\texttt{Method\_info}}$.

(Constant_pool)

$$cp\_name \ R_{\texttt{Constant\_pool}} \langle cp\_tok, smp\rangle \iff cp\_name \ R_{\texttt{CP\_index} \to \texttt{CP\_info}} \ cp\_tok \ \wedge \ cp\_name \ R_{\texttt{SMP}} \ smp$$

It is in $R_{\texttt{CP\_index}} : \texttt{Class\_name} \times \texttt{Index} \leftrightarrow \texttt{Package\_tok} \times \texttt{Index}$ that the reorganisation is expressed essentially.

(Pack_methods) The method item and method component contain the implementations of both static and virtual methods.

$$\begin{aligned}
&methods_name \ R_{\texttt{Pack_methods}} \ method_comp \iff \\
&\forall \langle c_name, sig\rangle \ R_{\texttt{Method_ref}} \ \langle p_tok, c_tok, m_tok\rangle . \\
&\qquad methods_name(c_name, sig) \ R_{\texttt{Method_info}} \ (methods_comp.methods)(\texttt{method_offset}(p_tok, c_tok, m_tok))
\end{aligned}$$

(Pack_fields) In the class file, all the field information is stored in the fields item. In the CAP file it is split between the static field and descriptor components. The former contains the values of the static fields, whereas the latter contains the flag and type information, for all fields. In the operational semantics we only use the type information, however.

As for the other components of the CAP file, we relate them to an aggregate, $fields\_name : \texttt{Class\_name} \rightarrow \texttt{Fields\_item}$, which groups all the fields items in class files of a package. There are four clauses to the definition: for values, instance fields, public static fields and package static fields.

$$
\begin{aligned}
&fields\_name\ R_{\texttt{Pack\_fields}}\ \langle Desc\_comp, Static\_field\_comp\rangle \iff \\
&\\
&\forall \langle c\_name, f\_name\rangle\ R_{\texttt{Field\_ref}}\ f\_ref\ . \\
&fields\_name\ c\_name\ f\_name\ .Field\_flags(\texttt{Static}) \Rightarrow \\
&fields\_name\ c\_name\ f\_name\ .Value = \\
&\qquad Static\_field\_comp\,.Non\_default\_values(\texttt{static\_field\_offset}(f\_ref)) \\
&\wedge \\
&\forall \langle c\_name, f\_name\rangle\ R_{\texttt{Instance\_field\_ref}}\ \langle c\_ref, f\_tok\rangle\ . \\
&\langle fields\_name\ c\_name\ f\_name, \langle c\_ref, f\_tok\rangle\rangle \\
&\qquad\qquad R_{\texttt{Instance\_field\_info}}\ Desc\_comp(c\_ref)\,.Fields\,.Instance\_fields(f\_tok) \\
&\wedge \\
&\forall \langle c\_name, f\_name\rangle\ R_{\texttt{Ext\_static\_field\_ref}}\ \langle c\_ref, f\_tok\rangle\ . \\
&\langle fields\_name\ c\_name\ f\_name, \langle c\_ref, f\_tok\rangle\rangle \\
&\qquad\qquad R_{\texttt{Pub\_static\_field\_info}}\ Desc\_comp(c\_ref)\,.Fields\,.Pub\_static\_fields(f\_tok) \\
&\wedge \\
&\forall \langle c\_name, f\_name\rangle\ R_{\texttt{Field\_ref}}\ f\_ref\ . \\
&\texttt{pack\_static}(c\_name, f\_name) \Rightarrow fields\_name\ c\_name\ f\_name \\
&\qquad\qquad R_{\texttt{Pack\_static\_field\_info}}\ Desc\_comp(c\_ref)\,.Fields\,.Pack\_static\_fields(f\_ref)
\end{aligned}
$$

We could have used $R_{\texttt{Field\_ref}}$ in each of the four clauses, but give the more specific relations when possible. For the package visible static methods we have no choice but to use the more general relation.

We now define the relations for the field information structures:

$$
\begin{aligned}
&\langle\langle flags, type, \_\rangle, f\_ref\rangle\ R_{\texttt{Instance\_field\_info}}\ \langle f\_tok, flags', f\_ref, type'\rangle \\
&\qquad \iff flags\ R_{\texttt{Field\_flags}}\ flags' \ \wedge\ type\ R_{\texttt{Type}}\ type'
\end{aligned}
$$

$$
\begin{aligned}
&\langle\langle flags, type, value\rangle, f\_ref\rangle\ R_{\texttt{Pub\_static\_field\_info}}\ \langle f\_tok, flags', f\_ref, type'\rangle \\
&\qquad \iff flags\ R_{\texttt{Field\_flags}}\ flags' \ \wedge\ type\ R_{\texttt{Type}}\ type'
\end{aligned}
$$

$$
\begin{aligned}
&\langle flags, type, value\rangle\ R_{\texttt{Pack\_static\_field\_info}}\ \langle flags', type'\rangle \\
&\qquad \iff flags\ R_{\texttt{Field\_flags}}\ flags' \ \wedge\ type\ R_{\texttt{Type}}\ type'
\end{aligned}
$$

(Class) We define $R_{\texttt{Class}}$. There are a number of equivalences expressing correctness of the construction of the class component. For the lookup, the significant ones are those between the method tables. These say that if a method is defined in the name format, then it must be defined (and equivalent) in the token format. Since the converse is not required, this means we can copy method tokens from a superclass. Instead, there is a condition saying that if there is a method token, then there must be a corresponding signature in some superclass.

If a method is visible in a class, then there must be an entry in the method table, indicating how to find the method information structure in the appropriate method component. For package visible methods this implies that the method must be in the same package. For public methods, if the two classes are in the same package, then this entry is an offset into the method component of this package. Otherwise, the entry is `0xFFFF`, indicating that we must use the method token to look in another package.

The class component only contains part of the information contained in the class files.

The full definition is (writing c_name for $cf.Class\_name$ and c_ref for $ci.Class\_ref$):

$$
\begin{array}{l}
cf : \mathtt{Class\_file}\ R_{\mathtt{Class}}\ ci : \mathtt{Class\_info} \iff \\
\left\{
\begin{array}{l}
cf.Class\_flags\ R_{\mathtt{Class\_flags}}\ ci.Class\_flags\ \wedge \\
cf.Super\ R_{\mathtt{Class\_ref}}\ ci.Super\ \wedge \\
\\
\forall sig \in cf.methods\_item. \\
public(sig) \Rightarrow \\
\exists m\_tok\ .\ public\_base \leq m\_tok < public\_base + public\_size\ \wedge \\
\langle c\_name, sig\rangle\ R_{\mathtt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle\ \wedge\ ci.Public\_table[m\_tok - ci.public\_base] = \\
\mathtt{method\_offset}(c\_ref, m\_tok) \\
\wedge \\
package(sig) \Rightarrow \\
\exists m\_tok\ .\ package\_base \leq m\_tok\ \&\ 127 < package\_base + package\_size\ \wedge \\
\langle c\_name, sig\rangle\ R_{\mathtt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle\ \wedge \\
ci.Package\_table[m\_tok\ \&\ 127 - ci.Package\_base] = \mathtt{method\_offset}(c\_ref, m\_tok) \\
\wedge \\
\forall m\_tok \in ci.Tables. \exists sig. \exists c'\_name. \\
\langle c'\_name, sig\rangle\ R_{\mathtt{Virtual\_method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge c\_name \leq c'\_name\ \wedge \\
public(sig) \Rightarrow \\
\qquad (same\_package(c\_name, c'\_name) \iff ci.public\_table[m\_tok - ci.public\_base] \neq \mathtt{0xFFFF})
\end{array}
\right.
\end{array}
$$

Finally, we define $R_{\mathtt{Package}}$. Recall that

$$[\![\mathtt{Package}]\!]_{name} = \mathtt{Class\_name} \to \mathtt{Class\_file}$$

$$[\![\mathtt{Package}]\!]_{tok} = \mathtt{CAP\_file}$$

We use the notations $\mathtt{class\_info}'(p\_tok, c\_tok)$ to indicate the class info for c_tok in the class component of CAP file p_tok (see p. 20), and $pack\_methods(p\_name)$ for the set of `Method` data extracted from each class file; similarly for $pack\_cps(p\_name)$ and $pack\_fields(p\_name)$.

$$\mathtt{pack\_cps} : \mathtt{Package} \to \mathtt{Constant\_pool}$$

$$\mathtt{pack\_cps}(p\_name) = \lambda\langle c\_name, i\rangle\ .\ p\_name(c\_name)\ .\ Constant\_pool\_item(i)$$

$$\mathtt{pack\_methods} : \mathtt{Package} \to \mathtt{Pack\_methods}$$

$$\mathtt{pack\_methods}(p\_name) = \lambda c\_name\ .\ p\_name(c\_name)\ .\ Methods\_item$$

$$\mathtt{pack\_fields} : \mathtt{Package} \to \mathtt{Fields}$$

$$\mathtt{pack\_fields}(p\_name) = \lambda c\_name\ .\ p\_name(c\_name)\ .\ Fields\_item$$

$$
\begin{array}{l}
p\_name\ R_{\mathtt{Package}}\ p\_tok \iff \\
\left\{
\begin{array}{l}
\mathtt{pack\_cps}(p\_name)\ R_{\mathtt{Constant\_pool}}\ p\_tok.cp\_comp\ \wedge \\
\forall c\_name\ R_{\mathtt{Class\_ref}}\ c\_ref\ .\ c\_name \in dom(p\_name) \Rightarrow p\_name(c\_name)\ R_{\mathtt{Class}}\ \mathtt{class\_info}'(p\_tok, c\_ref)\ \wedge \\
\mathtt{pack\_methods}(p\_name)\ R_{\mathtt{Pack\_methods}}\ p\_tok.method\_comp\ \wedge \\
\mathtt{pack\_fields}(p\_name)\ R_{\mathtt{Pack\_fields}}\ \langle p\_tok.desc\_comp, p\_tok.static\_field\_comp\rangle
\end{array}
\right.
\end{array}
$$

The offset functions link the various relations. We make a global assumption (in fact, local to an environment) of the existence of:

$$\mathtt{class\_offset}, \mathtt{method\_offset}, \mathtt{static\_field\_offset}$$

9 Proof

We first prove that the auxiliary functions preserve the appropriate relations.[9] Since we have not make the heap and environment explicit arguments, we need to assume the corresponding entities are related. Note that this proof is dependent on the specific implementations of auxiliary functions and, in particular, the choice of lookup algorithm used here.

Lemma 9.1 *If the heap and environment are related in the two formats, then: for all auxiliary functions $f : \theta \to \theta'$, given the corresponding preconditions, we have* $[\![f]\!]_{name}\ R_{\theta \to \theta'}\ [\![f]\!]_{tok}$

Proof: We will consider two cases.

($\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \to \texttt{Class\_ref} \times \texttt{Bytecode}$)

For `lookup`, this is achieved by inducting over the class hierarchy and using the constraints on $R_{\texttt{Class}}$ and $R_{\texttt{Method}}$. Formally, we prove that

$$\forall act\_name\ R_{\texttt{Class\_ref}}\ act\_ref\ .\ \forall (dec\_name, sig)\ R_{\texttt{Method\_ref}}\ (tag, dec\_ref, m\_tok).$$
$$\texttt{lookup\_name}(act\_name, (dec\_name, sig))\ R_{\texttt{Class\_ref}\times\texttt{Bytecode}}\ \texttt{lookup\_tok}(act\_ref, (m\_tok, dec\_ref))$$

Induction over classes is possible since the subclass ordering is well-founded.

If the reference is to a virtual method, then:

The functions `lookup_name` and `lookup_tok` have similar structures. `lookup_name` takes one of three branches and we show that the conditions and the results are equivalent for `lookup_tok`. Either the method is *defined and visible* in the actual class, or *defined and not visible*, or *undefined*.

- Suppose the method is defined and visible in the actual class, that is, $methods\_item(sig)$ is defined and the visibility condition holds.

  If the method token is public, then it must be that $m\_tok \geq public\_base$ and the offset is not `0xFFFF`.

  If the method token is package visible, then it must be greater than the package base, and the packages must be the same.

  In both cases, we return the actual class together with the code at that class.

  Now, by $R_{\texttt{Class}}$ we have that there exists a token m'_tok such that $\langle act\_name, sig\rangle\ R_{\texttt{Virtual\_method\_ref}}\ \langle act\_ref, m'\_tok\rangle$.

  Using $act\_name\ R_{\texttt{Class\_ref}}\ act\_ref$ and $(dec\_name, sig)\ R_{\texttt{Virtual\_method\_ref}}\ (dec\_ref, m\_tok)$, and the fact that dec_name and act_name are in the same hierarchy, we deduce that $(act\_name, sig)\ R_{\texttt{Virtual\_method\_ref}}\ (act\_ref, m\_tok)$. Thus, again by $R_{\texttt{Class}}$, it must be that $\texttt{method\_offset}(act\_ref, m\_tok)$ is the entry in the method table computed by the lookup.

  Then, by $R_{\texttt{Pack\_methods}}$, we get that the corresponding method information structures are related by $R_{\texttt{Method\_info}}$, and so in particular, the bytecodes are equivalent.

- Suppose the method is defined but not visible. This must be for a package token then, and we have $method\_tok \geq package\_base$ and the `same_package` condition is false.

  In both formats then we look at the superclass, which is the same due to the definition of $R_{\texttt{Class}}$, and because the environments and actual class references are related. Equality follows from the inductive hypothesis at the superclass.

-
  - If the function is not defined at the actual class in either format, then both algorithms look in the superclass and we appeal to the inductive hypothesis at the superclass. By the second constraint on $R_{\texttt{Virtual\_method\_ref}}$, this must be because the token is less than the base.
  - In the case where the two functions differ, that is, the method is undefined in the name format, but defined (and visible) in the token format, this must be because the method was copied from a superclass (and the token is greater than the base). We can then use the inductive hypothesis at this superclass, as in the previous case. This tells us that the results are equal at the superclass. Thus, by definition of `lookup_name` and the overriding constraint on $R_{\texttt{Virtual\_method\_ref}}$, we have that the results are equal in the current class.

[9] Often this is taken as part of the definition of a logical relation.

($\texttt{constant\_pool} : \texttt{Class\_ref} \to \texttt{Constant\_pool}$)

This follows directly from the assumption that the environments are related in the two formats and places restrictions on the definitions of $R_{\texttt{CP\_index}}$, $R_{\texttt{SM\_index}}$ and $R_{\texttt{CP\_info}}$. Commutation ensures that the constant pools are joined together without loss of data.

■

In order to use the operational semantics with the logical relations approach it is convenient to view the operational semantics as giving an interpretation. We define $[\![code]\!](\langle env, heap, op\_stack, loc\_vars, m\_ref\rangle)$ as the resulting state from the (unique) transition from $\langle code, op\_stack, loc\_vars\rangle$ with environment env and heap $heap$.

Thus we regard interpreted bytecode as having the type

$$\texttt{State} \to \texttt{Bytecode} \times \texttt{State}$$

where

$$\texttt{State} = \texttt{Global\_state} \times \texttt{Local\_state}$$

$$\texttt{Global\_state} = \texttt{Environment} \times \texttt{Heap}$$

$$\texttt{Local\_state} = \texttt{Operand\_stack} \times \texttt{Local\_variables} \times \texttt{Class\_ref}$$

Now, the following fact is trivial to show: if $R_B = id_B$ for all basic observable types, then $R_\theta = id_\theta$ for all observable θ. In combination with the following theorem, then, this says that if a transformation satisfies certain constraints (expressed by saying that it is contained in R) then it is correct, in the sense that no difference can be observed in the two semantics. In particular, we can observe the operand stack (of observable type $\texttt{Word}^*$) and the local variables (of observable type $\texttt{Nat} \to \texttt{Word}$) so these are identical under the two formats.

Theorem 9.2 *If*

1. $env_{name}\ R_{\texttt{Environment}}\ env_{tok}$,
2. $heap_{name}\ R_{\texttt{Heap}}\ heap_{tok}$,
3. $ls\ R_{\texttt{Local\_state}}\ ls'$, *and*
4. $code\ R_{\texttt{Bytecode}}\ code'$

then

$$[\![code]\!]_{name}(env_{name}, heap_{name}, ls)\ R_{\texttt{Bytecode}\times\texttt{State}}\ [\![code']\!]_{tok}(env_{tok}, heap_{tok}, ls')$$

Proof: It is straightforward to show that the representation independence of instructions follows from that of the auxiliary functions. Most of the work was in formulating the operational semantics so as to be independent of the underlying format.

We take `invokevirtual` as an example. We use subscripts to distinguish the interpretations in the two models. Suppose $heap_{name}\ R_{\texttt{Heap}}\ heap_{tok}$, $env_{name}\ R_{\texttt{Environment}}\ env_{tok}$, $m_{name}\ R_{\texttt{Method}}\ m_{tok}$. Then, by induction on `constant_pool`, we have $dec\_mref_{name}\ R_{\texttt{Method\_ref}}\ dec\_mref_{tok}$. By the assumption on $heap$ we have $act\_cref_{name}\ R_{\texttt{Class\_ref}}\ act\_cref_{tok}$. Thus, by induction on `lookup`, we get that $m\_class_{name}\ R_{\texttt{Class\_ref}}\ m\_class_{tok}$ and $m\_code_{name}\ R_{\texttt{Bytecode}}\ m\_code_{tok}$. Since the heap and environment do not change, we can conclude that `invokevirtual` is representation independent. The cases of the other instructions are proven similarly. ■

10 Conclusion

We have formalised the virtual machines and file formats for Java and Java Card, and the optimisation as a relation between the two. Correctness of this optimisation was expressed in terms of observable equivalence of the operational semantics, and this was deduced from the constraints that define the optimisation. Although the framework we have presented is quite general, the proof is specific to the instantiations of auxiliary functions we chose. It could be argued, in particular, that we might have proven the equivalence of two incorrect implementations of lookup. The remedy for this would be to specify the functions themselves, and independently prove their correctness.

It might seem that there is a circularity in the proof since the various relations make use of the environments, the equivalence of which is tantamount to the correctness we seek to prove. However, although the relations are defined using the environment, they do not use the relation between the environments. Moreover, the equivalence of environments does not assume the commutation of auxiliary functions but, rather, the equivalence of data that conforms to the specification.

There are a number of points, however, which are not entirely clear at this stage. We assumed functions to convert external tokens into internal offsets but it is not clear how they should be implemented. In this report we use the descriptor component to resolve tokens, but it may be that the export component should be used.

Also, it is not clear how array references are treated in the CAP file. In the constant pool in class file format, an array reference is a special form of class name, but this does not seem to be the case for class tokens, so have assumed a primitive function to convert an array as class name, into an array type.

For the semantics of the `new` instruction, we used a function `instance_fields` to compute the default values for the object fields. This requires the types of the fields, which are given in the descriptor component. Perhaps, though, it would suffice to use the `declared_instance_size` item in the corresponding class information structure.

Another unclear point is where the values of static fields are stored. We have assumed that they are stored in the field information structures of the fields item (in the class files) and the static field component (in the CAP files). This point is treated differently by [Ber97] and [BS98], and seemingly not considered by [Pus98].

The converter should produce an export file [Sun99] along with the CAP file but the details are not clear and we have not considered this. Finally, interface method references do not seem to appear from Draft 2 onwards of the JCVM 2.1 specification, but we have retained them here.

In addition to these problems, we have made a number of simplifications which could be relaxed. First of all, the proof should be extended to account for the rest of the transformation, accounting for interfaces, exceptions, and so on. It would also be easy to incorporate AID's and so make package tokens internal. Another extension would be to incorporate the export files and descriptor component. We found it more convenient to formalise various structures as functions where, in reality, they are actually laid out as tables. We could envisage another transformation pass where the functions are 'flattened' into tables.

We have used a simple definition of $R_{\texttt{Bytecode}}$ here, which just accounts for the changing indexes into constant pools (as well as method references in configurations). We have not considered inlining or the specialisation of instructions, however. We expressed equivalence in terms of an identity at observable types but, more realistically, we should account for the difference in word size. This has been considered in [LR98]. Although it seems that 'conversion' and 'optimisation', to borrow their terminology, are orthogonal, it would, nevertheless, be interesting to extend our formalisation to include these aspects. Although the specialisation of instructions could be handled by our technique (suitably combined with a type analysis), the extension is not clear for the more non-local optimisations.

We emphasised that the particular form of operational semantics used here is orthogonal to the rest of the proof. This version suffices for the instructions considered here, but could easily be changed (along with the definition of $R_{\texttt{Bytecode}}$).

These definitions have been formalised in Coq, and the lemmas verified [Seg99]. The discipline this imposed on the work presented here was very helpful in revealing errors. Even just getting the definitions to type-check uncovered many errors. It is worth reflecting on the fact that Sun presents their specification as a formal definition of Java Card, which we have 'formalised' here, and then used as the basis of a formalisation in Coq!

We take the complexity of the proofs (in Coq) as evidence for the merit in separating the correctness of a particular algorithm from the correctness of the specification. In fact, the operational semantics, correctness of the specification, and development of the algorithm are all largely independent of each other.

As mentioned in the introduction, there are two main steps to showing correctness:

1. Give an abstract characterisation of all possible transformations and show that the abstract properties guarantee correctness.

2. Show that an algorithm implementing such a transformation exists.

We are currently working on a formal development of a tokenisation algorithm using Coq's program extraction mechanism together with constraint-solving tactics. The development is orthogonal to the proofs here and, in particular, independent of the formalisation of the bytecode semantics.

One detail that is important for the development of the algorithm is the domains of the various functions and relations. We have not been too precise about the domains of the partial functions, and have used a notion of relational bijection accordingly. In fact, it suffices to think of the relations as being between *all* names and an infinite set of tokens but, in reality, we should use the *actual* names.

One significant improvement that could be made to the formalisation would be to use dependent types. This would offer a number of advantages. For example, if `CP_index` depended on a class reference, then we could avoid the explicit labelling of this. This would require an extension of the definition of logical relations, however.

In general, there are a number of changes which could be envisaged for the extraction. However, it was decided to 'freeze' the formalisation more or less in the form presented here so as to have a relatively stable version for the formalisation in Coq [Seg99].

References

[Ber97] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.

[BS98] E. Börger and W. Schulte. Defining the Java virtual machine as platform for provably correct Java compilation. In L. Brim, J. Grunska, and J. Zlatusla, editors, *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science*, volume 1450 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[LR98] Jean-Louis Lanet and Antoine Requet. Formal proof of smart card applets correctness. In *Third Smart Card Research and Advanced Application Conference (CARDIS'98)*, 1998.

[LY97] T. Lindholm and F. Yelling. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Mit96] J. Mitchell. *Foundations for Programming Languages*. Foundations of Computing Series. MIT Press, 1996.

[Mos98] Peter D. Mosses. Modularity in structural operational semantics. Extended abstract, November 1998.

[Pus96] Cornelia Pusch. Verification of Compiler Correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, pages 347–362. Springer-Verlag, 1996.

[Pus98] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.

[Seg99] Gaëlle Segouat. Preuve en Coq d'une mise en oeuvre de Java Card. Master's thesis, Projet Lande, IRISA, 1999.

[Sun97] Sun Microsystems. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997. Final Revision.

[Sun99] Sun Microsystems. *Java Card 2.1 Virtual Machine Specification*, March 1999. Final Revision 1.0.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399